НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Комп'ютерні системи та мережі»

спеціальності 123 «Комп'ютерна інженерія»

на тему: «Модуль для обчислення тригонометричних функцій з плаваючою комою»

Виконав:
студент IV курсу, групи ІО-61
Петрик Роман Ігорович _____

Керівник:
Професор, доктор технічних наук
Сергієнко Анатолій Михайлович _____

Консультант з нормоконтролю:
Професор, доктор технічних наук
Сімоненко Валерій Павлович _____

Рецензент:
Доцент кафедри СП і СКС, к.т.н.
Орлова Марія Миколаївна _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
«___»_____20__ р.

ЗАВДАННЯ
на дипломний проєкт студенту
Петрику Роману Ігоровичу

1. Тема проєкту «Модуль для обчислення тригонометричних функцій з плаваючою комою», керівник проєкту доцент, доктор технічних наук, Сергієнко Анатолій Михайлович, затверджені наказом по університету від «07» травня 2020 р. № 1081-с

2. Термін подання студентом проєкту _____

3. Вихідні дані до проєкту: розробити модуль для обчислення функцій синуса та косинуса в конвеєрному режимі з одинарною точністю вхідних та вихідних даних.

4. Зміст пояснювальної записки:

- Теоретичні засади;

- Опис алгоритму;

- Проектування модуля;

- Тестування модуля.

5. Перелік графічного матеріалу:

- Перетворення числа з фіксованою комою в число з плаваючою комою. Блок-схема алгоритму;

- Конвеєризований CORDIC. Схема електрична функціональна;

- Модуль для обчислення тригонометричних функцій з плаваючою комою. Схема електрична структурна.

6. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів дипломного проекту	Строк виконання етапів проекту	Примітка
1	*Затвердження теми роботи*	*10.12.2019*	*виконано*
2	*Огляд літератури та інших джерел з інформацією на дану тему*	*10.12.2019-01.02.2020*	*виконано*
3	*Аналіз існуючих рішень*	*01.02.2020-01.03.2020*	*виконано*
4	*Формування варіанту для вирішення завдання*	*01.03.2020-15.04.2020*	*виконано*
5	*Формування та реалізація алгоритму*	*15.04.2020-15.05.2020*	*виконано*
6	*Розробка та відлагодження програмного забезпечення*	*15.05.2020-20.05.2020*	*виконано*
7	*Оформлення матеріалів роботи*	*20.05.2020-24.05.2020*	*виконано*
8	*Передзахист*	*25.05.2020*	*виконано*
9	*Захист*	*15.06.2020*	*виконано*

Студент Петрик Роман Ігорович

Керівник Сергієнко Анатолій Михайлович

Анотація

Робота присвячена розробці пристрою для обчислення функцій синуса та косинуса кута в радіанах в конвеєрному режимі та одинарною точністю вхідних і вихідних даних. Пристрій на базі алгоритму для апаратно-ефективної реалізації CORDIC описується мовою VHDL. При цьому оптимізується процес обчислення та дані представляються у форматі з плаваючою комою.

Аннотация

Работа посвящена разработке устройства для вычисления функций синуса и косинуса угла в радианах в конвейерном режиме и одинарной точностью входных и выходных данных. Устройство на базе алгоритма для аппаратно-эффективной реализации CORDIC описывается на языке VHDL. При этом оптимизируется процесс вычисления и данные представляются в формате с плавающей запятой.

Abstract

The work is devoted to the development of a device for calculating the functions of sine and cosine of the angle in radians in the conveyor mode and the single-precision format of input and output data. The device based on the algorithm for hardware-efficient implementation of CORDIC is described in VHDL. This optimizes the calculation process and presents the data in a floating-point format.

№ рядка	формат	Позначення	Найменування	кількість	Примітка
			Документація загальна		
			Розроблена заново		
	А4	ІАЛЦ.467400.001 ВП	Відомість дипломного	1	
			проєкту		
	А4	ІАЛЦ.467400.002 ТЗ	Технічне завдання	4	
	А4	ІАЛЦ.467400.003 ПЗ	Пояснювальна записка	62	
	А4	ІАЛЦ.467400.004 Д1	Перетворення числа з	1	
			фіксованою комою у число		
			з плаваючою комою.		
			Блок-схема алгоритму		
	А4	ІАЛЦ.467400.005 Д2	Конвеєризований CORDIC.	1	
			Схема електрична		
			функціональна		
	А4	ІАЛЦ.467400.006 Д3	Модуль для обчислення	1	
			тригонометричних функцій		
			з плаваючою комою.		
			Схема електрична		
			структурна		

					ІАЛЦ.467400.001 ВП
Зм.	Арк.	№ докум.	Підпис	Дата	
Розроб.		Петрик Р.І.			Модуль для обчислення тригонометричних функцій з плаваючою комою. Відомість проєкту
Перевір.		Сергієнко А.М.			
Н. контр.		Сімоненко В.П.			
Затв.		Стіренко С.Г.			

Аркуш	Аркушів
1	1

НТУУ "КПІ" ФІОТ ІО-61

Технічне завдання до дипломного проєкту

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання розповсюджується на розробку модуля для обчислення тригонометричних функцій з плаваючою комою.

Область застосування: розробка цифрових пристроїв.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки служить завдання на виконання розробки модуля для обчислення тригонометричних функцій з плаваючою комою, затвердженого кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний Інститут ім. Ігоря Сікорського».

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проєкту є розробка модуля для обчислення тригонометричних функцій з плаваючою комою.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелами для розробки служать науково-технічна література та публікації в Інтернеті, що стосуються даної теми.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Склад виробу

Виробом є віртуальний модуль, який є конфігурованим і придатним до вбудови у ПЛІС довільної серії. Модуль повинен бути описаний мовою VHDL стилем для синтезу з використанням бібліотеки IEEE.

5.2. Технічні параметри

Віртуальний модуль повинен виконувати обчислення функцій sin(x) та cos(x) в конвеєрному режимі.

Віртуальний модуль повинен бути настроюваним. Настроюваними є передусім розрядність вхідних та вихідних даних.

Вхідні дані — числа з плаваючою комою у діапазоні [-π; π).

Вихідні дані — числа з плаваючою комою у діапазоні [-1;1].

Розрядність вхідних та вихідних даних — регульована під час компіляції і не перевищує 32.

Похибка обчислень повинна бути не більшою за вагу молодшого розряду результату.

Розрядність проміжних результатів вибирається під час розробки і повинна забезпечувати мінімальну похибку обчислень.

Серія цільової ПЛІС — Xilinx Virtex 6. Мінімальний період тактового інтервалу при конвеєрних обчисленнях — не більше 10 нс (уточнюється під час розробки).

5.3. Вимоги до надійності

Параметри надійності модуля визначаються надійністю цільової ПЛІС. Ці параметри забезпечуються за умов:

— модуль успішно протестований в розробленому стенді для іспитів;

— розроблений модуль успішно пройшов етапи синтезу, розміщенння і розведення у заданій мікросхемі ПЛІС;

— тривалість мінімального тактового інтервалу, яка розраховується у САПР ПЛІС для використання модуля у відповідних умовах (діапазон температур, напруга живлення), не перевищує задану;

— модуль вбудовується у систему з урахуванням усіх розроблених рекомендацій по його застосуванню. Строк використання модуля — необмежений.

5.4. Принцип роботи

Модуль повинен виконувати обчислення за відомим алгоритмом CORDIC. Обчислення повинні виконуватись у конвеєрному режимі, тобто,

при подачі вхідних даних чергою з періодом в один такт модуль повинен видавати результати у кожному такті, починаючи з n-го.

5.5. Умови експлуатації

Всі експлуатаційні вимоги співпадають з вимогами експлуатації мікросхем ПЛІС, для яких розробляється модуль.

5.6. Програмне забезпечення

Розробка модуля повинна проводитись з використанням симулятора VHDL, такого, як Active-HDL. Перевірка синтезу, розміщення та трасування повинні виконуватись у САПР Xilinx ISE ver. 14.

5.7. Вимоги до патентної чистоти

Структурні ознаки розробленого модуля не повинні попадати у межі дії охоронних обмежень, які закріплені патентами, що діють в Україні.

5.8. Спеціальні вимоги

Спеціальні вимоги до розробки не висуваються.

Пояснювальна записка
до дипломного проєкту

на тему: Модуль для обчислення тригонометричних функцій з плаваючою комою

Київ – 2020 року

ЗМІСТ

ВСТУП

У ландшафті цифрової обробки сигналів вже давно переважають мікропроцесори з вдосконаленнями, такими як одноциклічні інструкції з множенням-накопиченням та спеціальні режими адресації. Хоча ці процесори мають низьку вартість і пропонують надзвичайну гнучкість, вони часто недостатньо швидкі для справді складних завдань цифрових сигнальних процесорів. Поява реконфігурованих логічних комп'ютерів дозволяє підвищити швидкість спеціалізованих апаратних рішень за ціною, що є конкурентоспроможною традиційному програмному підходу. На жаль, алгоритми, оптимізовані для цих мікропроцесорних систем, зазвичай не відображаються в апаратному забезпеченні. Незважаючи на те, що апаратно-ефективні рішення часто існують, домінування програмних систем утримувало ці рішення поза увагою. Серед цих апаратно-ефективних алгоритмів – клас ітераційних рішень для тригонометричних та інших трансцендентних функцій, які використовують лише зсуви та додавання для обчислення. Тригонометричні функції засновані на векторних обертаннях, тоді як інші функції, такі як квадратний корінь, реалізуються за допомогою інкрементного вираження потрібної функції. Тригонометричний алгоритм називається CORDIC – абревіатура для coordinate rotation digital computer (цифровий комп’ютер обертання координат). Інкрементні функції виконуються з дуже простим розширенням до апаратної архітектури, і, хоча це не CORDIC в строгому розумінні, часто включаються через близьку схожість. Алгоритми CORDIC, як правило, виробляють один додатковий біт точності для кожної ітерації.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ЗАСАДИ

CORDIC - це абревіатура, придумана Джеком Е. Волдером для опису алгоритму цифрового комп'ютерного обертання координат, який він розробив у 1959 році [1]. У той час він використовувався для навігаційної системи в реальному часі і був додатково розширений Волдером у 1971 році [2]. Він використовується для швидкого обчислення елементарних функцій, таких як множення, ділення, тригонометричні функції, логарифмічна функція та різні перетворення, як перетворення прямокутної в полярну координату, і навпаки. Хоча CORDIC може бути не найшвидшою технікою виконання цих операцій, вона приваблива через простоту її апаратної реалізації, оскільки той самий ітераційний алгоритм може бути використаний для всіх цих додатків за допомогою основних операцій з додаванням зсувів. Алгоритм CORDIC може застосовуватися у двох режимах (режими обертання і векторизації) та трьох типах (лінійний, круговий і гіперболічний типи). Алгоритм є дуже привабливим для апаратної реалізації, оскільки він використовує лише елементарні операції зсуву та додавання для здійснення обертання вектора. Для цього потрібні лише 2 зсуви та 3 суматори, тому його розсіювання потужності дуже мале, а також він є дуже компактним. Він часто використовується в масиві обробних елементів на чіпах дуже великої степені інтеграції [3].

Алгоритми цифрової обробки сигналів (DSP) виявляють зростаючу потребу в ефективній реалізації складних арифметичних операцій. Обчислення тригонометричних функцій, перетворень координат або обертання складних фазових значень майже закономірно пов'язане із сучасними алгоритмами DSP. Популярні приклади застосування – алгоритми, що використовуються в цифровій технології зв'язку та в адаптивній обробці сигналів. Незважаючи на те, що в цифрових комунікаціях важлива прямолінійна оцінка цитованих функцій, численні алгоритми адаптивного оброблення сигналів на основі матриці вимагають рішення систем лінійних

рівнянь, QR-факторизації або обчислення власних значень, власних векторів або сингулярних значень [4]. Усі ці завдання можна ефективно реалізувати, використовуючи елементи обробки, які виконують векторні обертання. Алгоритм цифрового комп'ютера обертання координат (CORDIC) пропонує можливість обчислити всі бажані функції досить простим і елегантним способом.

Алгоритм може бути закодований як в мікропрограмі, так і в невеликих мікроконтролерах. З незначними модифікаціями в початкових умовах та таблицях даних основний алгоритм може множити, ділити, модулювати та обчислювати квадратні корені, гіперболічні функції, та інші. Саме його універсальність та простота роблять CORDIC кращою реалізацією математичних функцій на малих ручних калькуляторах. Він обчислює тригонометричні функції синуса, косинуса, величини та фази (арктангенс) до будь-якої бажаної точності. Він також може обчислити гіперболічні функції. CORDIC реалізований у кишенькових калькуляторах, таких як HP 35 від Hewlett Packard, і в арифметичних копроцесорах, таких як Intel 8087. Деякі автори пропонували використовувати процесори CORDIC для програм обробки сигналів, таких як фільтрація, сингулярне розкладання величини [5], для обробки зображень або для вирішення лінійних систем. Алгоритм CORDIC знайшов своє широке застосування в обчисленні швидкої трансформації Фур'є [6]. Сьогодні алгоритм CORDIC використовується в дизайні нейро-мереж дуже великої степені інтеграції, високоефективних додатках обертання векторів DSP, вдосконаленому дизайні схем, оптимізованому проектуванні з низькою потужністю [7].

1.1. Алгоритм CORDIC

Загальновідомі функції sin θ, cos θ, $\sin^{-1}$ θ, $\cos^{-1}$ θ, sinh θ, cosh θ, $\sinh^{-1}$ θ, $\cosh^{-1}$ θ, ex, log x тощо - всі члени важливого класу функцій з математики, відомих як елементарні функції. Елементарні функції унікальні тим, що їх обчислюють саме в кінцевій кількості арифметичних операцій - їх точне

подання вимагає використання нескінченного ряду алгебраїчних термінів. Однак їх можна наблизити до потрібної точності, використовуючи обмежену кількість операцій. Вони використовуються в таких різноманітних програмах, як робототехніка, 3-D комп'ютерна графіка, SVD декомпозиція, цифрова обробка сигналів. Обчислення тригонометричної функції - це трудомістка операція. Фактично з усіх арифметичних операцій, які повинен виконувати чіп, тригонометричні функції мають найгіршу затримку [8]. Для їх проведення потрібно багато циклів, щоб інструкції, що залежать від їх оцінки, повинні зупинятися, поки результат не стане доступним. Додатково такі ресурси, як суматори, перемикачі та блоки множення, пов'язані і недоступні для використання навіть за допомогою інших незалежних інструкцій, що призводить до зупинок через структурні небезпеки. Тому вкрай важливо, щоб ці елементарні функції були обчислені якомога швидше, щоб уникнути погіршення продуктивності. Результати повинні бути отримані з високою точністю для будь-якого з кутів у межах принципової області елементарної функції. Такі методи, як зменшення діапазону, корисні для відображення аргументу як його основне значення, але навіть тому алгоритм повинен бути достатньо ефективним, щоб надати точну відповідь з будь-якою вказаною точністю. Сьогодні енергоспоживання стало важливим показником в електронному дизайні, особливо, оскільки гаджети та обчислювальні пристрої зменшуються в розмірах. Тепло, яке розсіюється за рахунок споживання енергії в обчислювальній мікросхемі, робить чіп важким для охолодження. Ця мікросхема повинна працювати або при зниженому рівні продуктивності, щоб запобігти її згоряння, або ж для використання температури, щоб знизити температуру до рівня керованості, необхідно використовувати дорогі та об'ємні механізми охолодження, такі як радіатори або повітря. Коли елементи обчислювальної голодної енергії використовуються на споживчих пристроях, таких як цифрові камери або MP3-програвачі, вони швидко розряджають акумулятор, що призводить до поганого досвіду для користувача. Так чи інакше, надмірне споживання

електроенергії обмежує продуктивність арифметичної мікросхеми. Існує п'ять основних способів, за допомогою яких елементарна функція може бути обчислена апаратно - за допомогою пошуку таблиць, поліноміального наближення, раціонального наближення, методів CORDIC та квадратичної конвергенції. Метод CORDIC - це найбільш універсальний з усіх алгоритмів, який можна використовувати для оцінки елементарних функцій. Це ж обладнання можна використовувати для обчислення тригонометричних співвідношень (sin, cos, tan та ін.), Гіперболічних співвідношень (sinh, cosh, tanh), множення, ділення, зворотних тригонометричних (дуги, дуги) та зворотних гіперболічних співвідношень (arcsinh, arccosh). З невеликою модифікацією він також може обчислювати логарифми, експоненти тощо.

1.1.1. Вступ до алгоритму CORDIC

Алгоритм Волдера отриманий із загальних рівнянь обертання вектора. Волдер намагався вдосконалити системи навігації в режимі реального часу, використовувані в бомбардувальнику B-58 [9]. У ті часи аналогічні прилади використовувались для вирішення складних навігаційних рівнянь, які допомагали знаходити положення літака на землі. Однак вони володіли обмеженою точністю. Він обчислив синус і косинус кута, перемістивши вектор від його початкового положення (вздовж осі X) до його кінцевого положення, де він лежав похилим під деяким кутом 0 до осі X. Вектор переміщувався послідовно невеликими кроками, одночасно оновлюючи X і Y координати вектора після кожного кроку. Операцію оновлення було просто виконати. Як тільки вектор досяг свого цільового кутового положення, його кінцеві координати X і Y дали значення косину і синуса кута нахилу 0. Алгоритм CORDIC використовується для оцінки обчислення елементарних функцій в реальному часі за допомогою ітераційного обертання вводу вектор. Обертання заданого вектора реалізується за допомогою послідовності обертів з фіксованими кутами, що призводить до загального обертання через заданий кут або призводить до остаточного кутового аргументу нуля. Однак

головним недоліком алгоритму CORDIC є його повільна обчислювальна швидкість. Для ітеративної структури CORDIC швидкість виконання операції CORDIC обмежена великим числом ітерації N, яка, як правило, дорівнює внутрішній довжині слова, Вт. На алгоритмічному рівні одним тривіальним рішенням для подолання такої проблеми є зменшення числа ітерації безпосередньо . Було введено ряд методів для зменшення кількості ітерацій CORDIC для підвищення його продуктивності. Інтернет CORDIC був розроблений Ercegovac та Lang [10] для додатків, де вхідні біти стали доступні серійно. Онлайн-метод CORDIC замінює змінні перемикачі передач на більш ефективні затримки в області. Їх метод також міг компенсувати значення K в Інтернеті. Дупрат і Мюллер [3] беруть на себе скорочення часу циклу ітерації CORDIC за допомогою швидких суматорів, заснованих на використанні надмірної арифметики для вираження операндів. У разі керування CORDIC, дивергентні обертання повністю усуваються шляхом усунення прострілу [11]. Для застосувань, які потребують лише обертання вперед (або векторного обертання), техніка Angle Recoding забезпечує розслаблений підхід для прискорення роботи алгоритму CORDIC [12]. Алгоритм цифрового комп'ютерного обертання координат пропонує можливість обчислити всі бажані функції простим та елегантним способом. Зважаючи на вимоги та обмеження різних середовищ прикладних програм, розробка алгоритму та архітектури CORDIC відбулася для досягнення високої пропускної здатності та зменшення складності обладнання, а також затримки впровадження [13]. Деякі з типових підходів до впровадження зменшеної складності орієнтовані на мінімізацію складності операції масштабування та складності перемикача в двигуні CORDIC. Затримка реалізації - невід'ємний недолік звичайного алгоритму CORDIC. Для зменшення затримки були розроблені схеми кутового перекодування, обертання змішаного зерна та CORDIC з більшим радіусом. Паралельні та конвеєрні CORDIC були запропоновані для обчислення високої пропускної здатності [14].

1.2. Формат чисел

Формат чисел у комп'ютері - це внутрішнє подання числових значень у цифровому комп'ютерному апаратному та програмному забезпеченні. Зазвичай числові значення зберігаються як групування бітів, названі для кількості бітів, що їх складають. У реальному житті ми маємо справу з реальними числами - це числа з дробовою частиною. У більшості сучасних комп’ютерів у нас є апаратна підтримка фіксованих точок та номерів, що представляють собою точки для представлення реальних чисел.

Представлення числа з фіксованою точкою: Форматування з фіксованою точкою корисно для представлення дробів у двійковій формі. У поданні з фіксованою точкою кожне слово має однакову кількість цифр, а двійкова точка завжди фіксується на одній позиції. Впроваджуючи алгоритми, що використовують математику з фіксованою точкою, можна помітити значне поліпшення швидкості виконання через невід'ємну технічну підтримку математики з великою кількістю процесорів, а також зменшену складність програмного забезпечення для множення та поділу емуляційного цілого числа. Це підвищення швидкості відбувається за рахунок зменшеного діапазону та точності змінних алгоритму.

Формат $Q_{m.n}$: m біт для цілої частини, n біт для дробової частини.

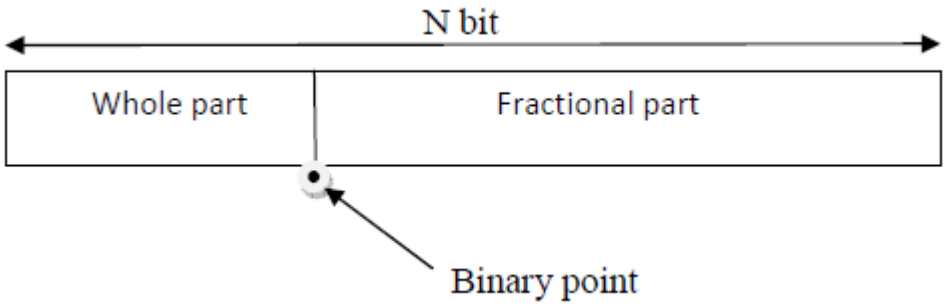


Рис. 1.1. Представлення числа з фіксованою комою

Представлення чисел з плаваючою комою: В обчисленні кома описує метод подання реальних чисел таким чином, що може підтримувати широкий діапазон значень. Числа, як правило, представлені приблизно до фіксованої

кількості значущих цифр і масштабуються за допомогою показника. Основою для масштабування зазвичай є 2, 10 або 16. Типове число, яке може бути точно представлено, має форму:

$$\text{Мантиса} \cdot \text{Основа}^{\text{Експонента}}$$

Термін «плаваюча кома» позначає той факт, що кома (десяткова кома або, що частіше буває у комп’ютерах, двійкова кома) може «опуститись», тобто вона може бути розміщена де завгодно відносно значущих цифр числа. Ця позиція окремо вказується у внутрішньому представництві, і таким чином подання «плаваючої коми» може розглядатися як комп'ютерна реалізація наукових позначень. Перевага уявлення про плаваючу кому полягає в тому, що воно може підтримувати набагато ширший діапазон значень і дуже точне.

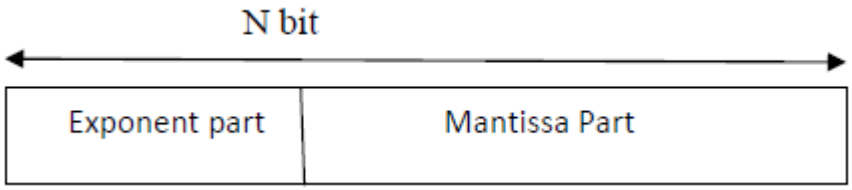


Рис. 1.2. Представлення числа з плаваючою комою

ВИСНОВКИ ДО РОЗДІЛУ 1

1. Після аналізу необхідних теоретичних засад було отримано уявлення про універсальність алгоритму для обчислення елементарних функцій CORDIC, його можливі недоліки, які полягають в затримках при обчислювальному процесі, а також які усуваться в оптимізованих версіях алгоритму.
2. Було отримано уявлення про відмінність між форматами подання чисел, а саме форматом з фіксованою комою та форматом з плаваючою комою, що дозволить встановити відношення між даними видами подання чисел у двійковій системі числення.

РОЗДІЛ 2
ОПИС АЛГОРИТМУ

Алгоритм CORDIC був вперше введений Волдером [1] для обчислення тригонометричних функцій, множення, ділення та перетворення типів даних, а пізніше узагальнений Волтером до гіперболічних функцій [2]. Алгоритм CORDIC не використовує методи, засновані на обчисленні, такі як поліноміальне чи раціональне наближення функції. CORDIC обертається навколо ідеї "обертання" фази складного числа шляхом множення на послідовність постійних значень. Однак множники можуть усі бути степенями двійки, тому в двійковій арифметиці їх можна виконати за допомогою лише зсувів та додавання, фактичний множник не потрібен порівняно з іншими підходами. CORDIC дає можливість виконувати обертання і, таким чином, обчислювати функції синуса, косинуса та арктангенсу, а також множити чи ділити числа, використовуючи лише елементарні операції зсуву та додавання. Тому CORDIC є явним переможцем, коли апаратний множник недоступний, наприклад в мікроконтролері, або коли ми хочемо зберегти ворота в ПЛІС. Алгоритми CORDIC вимагають лише зсувів, додавання та пошуку в таблиці, тобто простої цілочисельної математики. Таким чином, можна реалізувати спеціалізовану машину CORDIC, невелику і досить швидку для розрахунків у режимі реального часу, присвячену цій єдиній цілі.

Відомо два основні режими CORDIC, що ведуть до обчислення різних функцій: режим обертання та режим векторизації. Для обох режимів алгоритм може бути реалізований як ітеративна послідовність додавання або віднімання та операцій зсуву, які є обертаннями за фіксованим кутом повороту (іноді його називають мікро-обертанням), але зі змінним напрямком обертання. Завдяки простоті залучених операцій алгоритм CORDIC дуже добре підходить для реалізації VLSI.

2.1. Основні рівняння алгоритму CORDIC

Всі тригонометричні функції можна обчислити або вивести з функцій, використовуючи векторні обертання, як буде обговорено в наступних розділах. Вектор обертання також може використовуватися для полярних у прямокутні та прямокутні в полярні перетворення, для векторної величини та як будівельний блок у певних перетвореннях, таких як DFT та DCT. Алгоритм CORDIC надає ітераційний метод виконання векторних обертань довільними кутами, використовуючи лише зсуви та додавання. Алгоритм Волдера [4] отриманий із загальних рівнянь обертання вектора. Якщо вектор V з координатами (x, y) обернений на кут Ø, то новий вектор V’ може бути описаний координатами (x’, y’), де x’ і y’ можна виразити, використовуючи наступні залежності між x, y і Ø.

$$X = r\cos\theta,\ Y = r\sin\theta \tag{2.1}$$

$$V' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x\cos Ø - y\sin Ø \\ y\cos Ø + x\sin Ø \end{bmatrix} \tag{2.2}$$

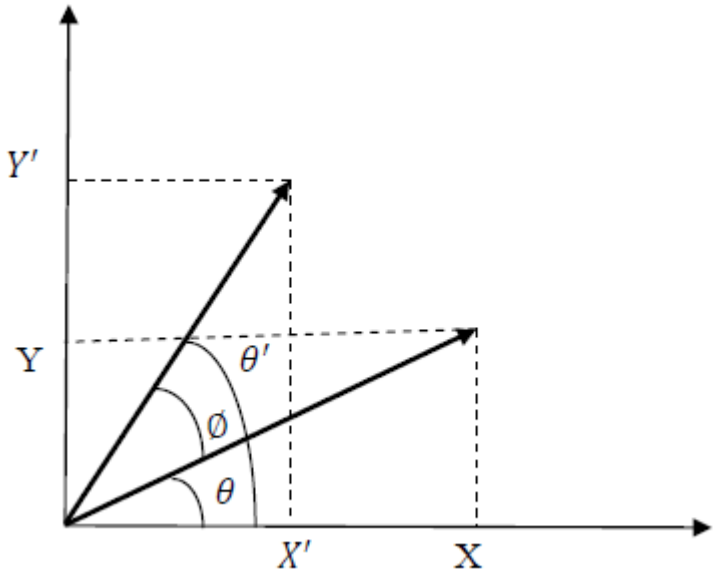


Рис. 2.1. Поворот вектора V на кут Ø

Знайдемо, як рівняння (2.1) та (2.2) виражаються на рис. 2.1. З рис. 2.1 вектор V(x, y) може бути розкладений на 2 частини вздовж осей x та y як r

cos θ та r sin θ відповідно. Рис. 2.2 ілюструє обертання вектора V = $\begin{bmatrix} x \\ y \end{bmatrix}$ на кут θ.

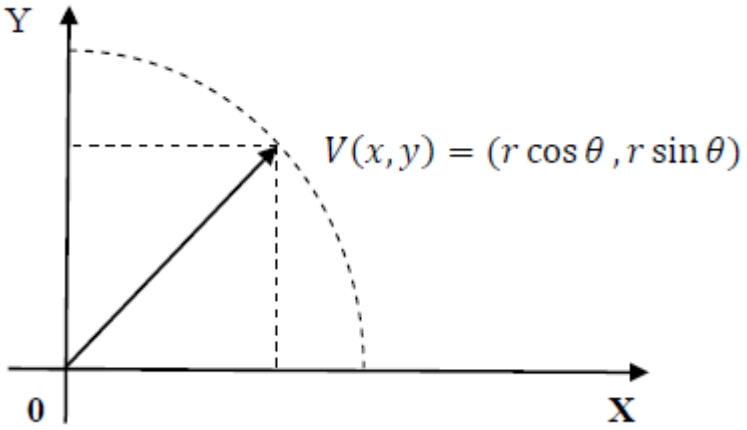


Рис. 2.2. Вектор V з величиною r та фазою θ

, звідки:

$$\left.\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \end{aligned}\right\} \tag{2.3}$$

Аналогічно з рис. 2.1 видно, що вектор V і V’ можна розкласти на дві частини. Нехай V має величину і фазу як r і θ відповідно, а V’ має величину і фазу як r і θ’, де V’ отриманий на рис. 2.1 після обертання вектора V проти годинникової стрілки на кут Ø. З рис. 2.1 це можна побачити:

$$\theta' - \theta = \emptyset \tag{2.4}$$

, звідки:

$$\theta' = \theta + \emptyset \tag{2.5}$$

$$\begin{aligned} OX' = x' &= r\cos\theta' \\ &= r\cos(\theta + \emptyset) \\ &= r(\cos\theta\cos\emptyset - \sin\theta\sin\emptyset) \\ &= (r\cos\theta)\cos\emptyset - (r\sin\theta)\sin\emptyset \end{aligned} \tag{2.6}$$

З рис. 2.2 та рівняння 2.6, OX’ може бути виражений як:

$$OX' = x' = x\cos\emptyset - y\sin\emptyset \tag{2.7}$$

Аналогічно:

$$\begin{aligned} OY' = y' &= r\sin\theta' \\ &= r\sin(\theta + \emptyset) \\ &= r(\sin\theta\cos\emptyset + \cos\theta\sin\emptyset) \\ &= (r\sin\theta)\cos\emptyset + (r\cos\theta)\sin\emptyset \\ &= y\cos\emptyset + x\sin\emptyset \end{aligned} \tag{2.8}$$

V’ після обертання проти годинникової стрілки на кут Ø:

$$x' = x\cos\emptyset - y\sin\emptyset$$

$$y' = y\cos\emptyset + x\sin\emptyset$$

Аналогічно, значення вектора V’ після обертання за годинниковою стрілкою вектора V на кут Ø:

$$x' = x\cos\emptyset + y\sin\emptyset \tag{2.9}$$

$$y' = y\cos\emptyset - x\sin\emptyset \tag{2.10}$$

Рівняння (2.7), (2.8), (2.9), (2.10) можуть бути виражені в формі матриці як:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\emptyset & \mp\sin\emptyset \\ \pm\sin\emptyset & \cos\emptyset \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.11}$$

V’ після обертання проти годинникової стрілки вектора V на кут Ø:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\emptyset & -\sin\emptyset \\ \sin\emptyset & \cos\emptyset \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Волдер [1] зауважив, що, розподіляючи cos Ø з обох сторін, отримане рівняння буде дотичне дотичного кута Ø, під кутом якого ми хочемо знайти значення синуса та косинуса. Далі, якщо вважати, що кут θ являє собою сукупність малих кутів, а складені кути вибирають таким чином, що їх дотичні значення мали всі зворотні степені двійки, то це рівняння [15] можна переписати у вигляді ітеративної формули:

$$x' = \cos \emptyset \, (x - y \tan \emptyset) \tag{2.12}$$

$$y' = \cos \emptyset \, (y + x \tan \emptyset) \tag{2.13}$$

$$z' = z + \emptyset$$

, де Ø – кут обертання (знак $\pm$ вказує на напрямок обернання), а z – кутовий акумулятор.

Множення на дотичний член можна уникнути, якщо кути повороту, i, отже, tg Ø, обмежені так, що tg $\emptyset = 2^{-i}$. У цифровому обладнанні це позначає просту операцію зсуву. Крім того, якщо ці обертання виконуються повторно, і в обох напрямках кожне значення tg Ø є репрезентабельним. З $\emptyset = tg^{-1}2^{-i}$ член косинуса також може бути спрощений, оскільки cos (Ø) = cos (-Ø) і це є константою для фіксованого числа ітерацій:

$$x' = \cos(\emptyset) \, (x - y. \, 2^{-i})$$
$$y' = \cos(\emptyset) \, (y + x. \, 2^{-i})$$

Ø може бути позитивним або негативним, залежно від напрямку обертання. Рівняння (2.11) може бути виражене як:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos \emptyset \begin{bmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.14}$$

Знову модифікація рівняння (2.14) як за годинниковою стрілкою, так і проти годинникової стрілки. Це можна виразити як:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos \emptyset \begin{bmatrix} 1 & -d_i 2^{-i} \\ d_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.15}$$

Ітеративне рівняння (2.15) може бути виражене як:

$$x_{i+1} = k_i (x_i - y_i d_i 2^{-i}) \tag{2.16}$$

$$y_{i+1} = k_i (y_i + x_i d_i 2^{-i}) \tag{2.17}$$

, де i позначає величину обертання, необхідного для досягнення шуканого кута шуканого вектора, $k_i = \cos (tg^{-1} (2^{-i}))$, а $d_i = \pm 1$, добуток k_i, являє собою так званий коефіцієнт K:

$$K = \prod_{i=0}^{n-1} k_i$$

, де $\prod_{i=0}^{i=n-1} k_i = \cos \emptyset_1 \cos \emptyset_2 \cos \emptyset_3 \cdots \cdots \cos \emptyset_{n-1}$ (тут Ø є кутом обертання для n обертань).

k_i – коефіцієнт наближення CORDIC. Для 16-бітного апаратного методу наближення CORDIC значення k_i :

$$k_i = \prod_{i=0}^{15} \cos \emptyset_i$$
$$= \cos \phi_0 \cos \phi_1 \cos \phi_2 \cos \phi_3 \cos \phi_4 \cos \phi_5 \cos \phi_6 \cos \phi_7$$
$$\cos \phi_8 \cos \phi_9 \cos \phi_{10} \cos \phi_{11} \cos \phi_{12} \cos \phi_{13} \cos \phi_{14} \cos \phi_{15}$$
$$= \cos 45^\circ \cos 26.565^\circ \cos 14.036^\circ \ldots\ldots\ldots \cos 0.00699^\circ \cos 0.00349^\circ \cos 0.00175^\circ$$
$$= 0.6073$$

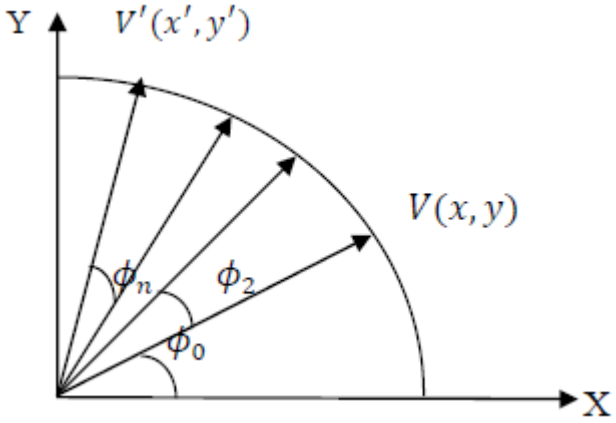


Рис. 2.3. Об’єднання декількох мікро-обертань

З рис. 2.3 кілька мікро-обертань можуть бути об’єднані при русі вектора дискретними кутовими кроками $(\phi_0, \phi_1, \phi_2, \ldots\ldots\ldots \phi_n)$ від початкового положення V (x, y) до свого цільового положення V'(x', y'):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos \phi_0 \cos \phi_1 \ldots \cos \phi_n \begin{bmatrix} 1 & -\tan \phi_0 \\ \tan \phi_0 & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & -\tan \phi_n \\ \tan \phi_n & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad (2.18)$$

Для 16-бітового методу CORDIC косинус і синус кута ф можна представити у матричній формі за допомогою рівняння (2.18):

$$\begin{bmatrix}\cos\phi \\ \sin\phi\end{bmatrix} = \begin{bmatrix}1 & -\tan\phi_0 \\ \tan\phi_0 & 1\end{bmatrix} \dots\dots\dots \begin{bmatrix}1 & -\tan\phi_{15} \\ \tan\phi_{15} & 1\end{bmatrix} \begin{bmatrix}0.6073 \\ 0\end{bmatrix} \quad (2.19)$$

Для 24-бітового методу CORDIC значення k_i :

$$k_i = \prod_{i=0}^{i=23} \cos\phi_i = \cos\phi_0 \cos\phi_1 \dots\dots\dots \cos\phi_{22} \cos\phi_{23} = 0.6073$$

Для 32-бітового методу CORDIC значення k_i :

$$k_i = \prod_{i=0}^{i=31} \cos\phi_i = \cos\phi_0 \cos\phi_1 \dots\dots \cos\phi_{30} \cos\phi_{31} = 0.6073$$

Наближення CORDIC k_i однакове для всіх апаратних методів CORDIC.

2.1.1. Псевдо-обертання

Псевдо-обертання отримується після відкидання косинуса. Псевдо-обертання отримується, виконуючи задане обертання: кут повороту правильний, але значення x і y масштабуються по cos (ф). Зауважимо, що це може зробити обчислення площин обертів більш схожими до простих операцій.

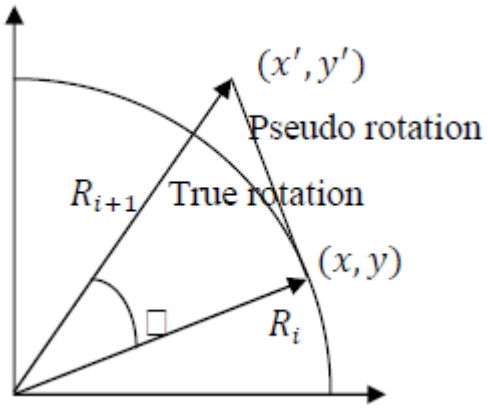


Рис. 2.4. Псевдо-обертання

Відкидаючи косинус, отримуємо:

$$x' = x - y \tan \phi \tag{2.20}$$

$$y' = y + x \tan \phi \tag{2.21}$$

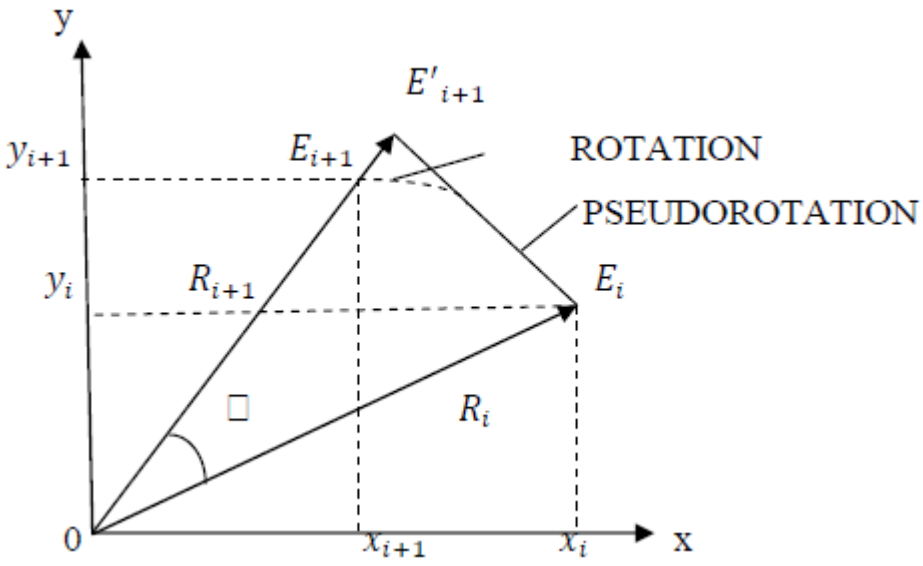


Рис. 2.5. Обертання та псевдо-обертання

Обертання вектора OE_i з кінцевою точкою в (x_i, y_i) на кут ф:

$$x_{i+1} = x_i \cos \phi_i - y_i \sin \phi_i$$
$$= \cos \phi_i \, (x_i - y_i \tan \phi_i)$$

$$= \frac{(x_i - y_i \tan \phi_i)}{(1 + \tan^2 \phi_i)^{1/2}}$$

$$y_{i+1} = y_i \cos \phi_i + x_i \sin \phi_i$$
$$= \cos \phi_i (y_i + x_i \tan \phi_i)$$
$$= \frac{(y_i + x_i \tan \phi_i)}{(1 + \tan^2 \phi_i)^{1/2}}$$

Тепер усунемо поділ на $(1 + tg^2 \varphi_i)^{1/2}$ і оберемо φ_i так, що tg φ_i - степінь двійки. В той час як справжнє обертання не змінює довжину R_i вектора, крок псевдо-обертання збільшує його довжину.

З рис. 2.5 можна побачити, що:

$$\frac{R_i}{R_{i+1}} = \cos\phi_i$$

$$R_{i+1} = \frac{R_i}{\cos\phi_i}$$

$$R_{i+1} = R_i(1 + \tan^2\phi_i)^{1/2}$$

Координати нової кінцевої точки E_{i+1} після псевдо-обертання отримують шляхом множення координат E_{i+1} на коефіцієнт розширення:

$$x_{i+1} = x_i - y_i \tan\phi_i \quad (2.22)$$

$$y_{i+1} = y_i + x_i \tan\phi_i \quad (2.23)$$

Оригінал заданого обертання тепер зводиться до алгоритму додавання ітеративного зсуву:

$$x_{i+1} = x_i - d_i y_i 2^{-i} \quad (2.24)$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \quad (2.25)$$

$$z_{i+1} = z_i - d_i \phi_i \quad (2.26)$$

$$d_i = \pm 1 \quad (d_i – напрямок обертання)$$

Кожна ітерація потребує:

- 2 зсуви;
- 1 перегляд таблиці;
- 3 додавання/віднімання.

2.1.2. Коефіцієнт масштабування

При спрощенні алгоритму [4] [15], щоб дозволити псевдо-обертання, косинус був відкинутий. Застосовуючи цю ітераційну схему, коефіцієнт масштабування K_n можна записати як:

$$K_n = \prod_{i=0}^{n-1}\left(\frac{1}{\cos\phi_i}\right) = \prod_{i=0}^{n-1}\sqrt{1+\tan^2\phi_i} = \prod_{i=0}^{n-1}\sqrt{1+2^{-2i}} = 1.6467$$

Так як кількість ітерацій n відома, значення K_n може бути попередньо обчислене.

Перемноживши рівняння (2.22) та (2.23) на cos φ_i , отримуємо:

$$\cos\phi_i \,.\, x_{i+1} = \cos\phi_i \,(x_i - y_i \tan\phi_i)$$
$$\cos\phi_i \,.\, y_{i+1} = \cos\phi_i \,(y_i + x_i \tan\phi_i)$$

, або:

$$\cos\phi_i \,.\, x_{i+1} = (x_i \cos\phi_i - y_i \sin\phi_i)$$
$$\cos\phi_i \,.\, y_{i+1} = (y_i \cos\phi_i + x_i \sin\phi_i)$$

Після n ітерацій маємо:

$$x_n = K_n(x_0 \cos\phi_i - y_0 \sin\phi_i) \quad (2.27)$$

$$y_n = K_n(y_0 \cos\phi_i + x_0 \sin\phi_i) \quad (2.28)$$

Для знаходження cos φ_i та sin φ_i при $x_0 = 1$ та $y_0 = 0$ маємо:

$$\frac{x_n}{K_n} = \cos\phi_i$$

$$\frac{y_n}{K_n} = \sin\phi_i$$

2.2. Основні ітерації алгоритму CORDIC

Основні ітерації:

$$x_{i+1} = x_i - d_i y_i 2^{-i} \quad (2.29)$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \quad (2.30)$$

$$z_{i+1} = z_i - d_i \phi_i \quad (2.31)$$

$$d_i = \pm 1$$

(d_i – напрямок обертання)

Обчислення x_{i+1} та y_{i+1} потребує i бітів зсуву та додавання/віднімання. Якщо функція $\varphi_i = tg^{-1}(2^{-i})$ була обчислена і знаходиться в таблиці для різних значень i, то для обчислення z_{i+1} достатньо одного додавання/віднімання.

Кожна ітерація CORDIC, таким чином, включає два зсуви, перегляд таблиці та три додавання/віднімання.

Якщо обертання виконується тим самим набором кутів (зі знаками + або -), тоді коефіцієнт розширення або коефіцієнт масштабування K_n постійний і може бути попередньо обчислений.

Ітерації CORDIC можна використовувати в двох режимах роботи. Ці два режими в основному відрізняються тим, як вибираються напрямки мікрообертання:

- Режим обертання;
- Векторний режим.

У режимі обертання [15] [16] кутовий акумулятор ініціалізується з потрібним кутом повороту ($z_0 = \theta$). Рішення обертання при кожній ітерації приймається для зменшення величини на основі знаку залишкового кута в кутовому акумуляторі. Отже, рішення на кожній ітерації базується на знаку залишкового кута після кожного кроку.

Для режиму обертання ітерації CORDIC наступні:

$$x_{i+1} = x_i - d_i y_i 2^{-i} \tag{2.32}$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \tag{2.33}$$

$$z_{i+1} = z_i - d_i \phi_i \tag{2.34}$$

$$d_i = \pm 1 \quad (d_i \text{ – напрямок обертання})$$

$$d_i = \begin{cases} +1, if\ z_i \geq 0 \\ -1, if\ z_i < 0 \end{cases} \tag{2.35}$$

Після n ітерацій в режимі CORDIC z_n досить близький до нуля, маємо:

$$z = \sum \phi_i \tag{2.36}$$

Вихідні дані режиму обертання x_n, y_n і z_n задаються наступними виразами, x_n і y_n є координатами вектора після обертання:

$$x_n = K_n(x_0 \cos z_0 - y_0 \sin z_0) \tag{2.37}$$

$$y_n = K_n(y_0 \cos z_0 + x_0 \sin z_0) \quad (2.38)$$

$$z_n = 0 \quad (2.39)$$

, де $K_n = \prod_{i=0}^{n-1}(1 + 2^{-2i})^{1/2} = 1.6467$.

2.2.1. Обчислення синуса та косинуса за методом CORDIC

Режим обертання алгоритму CORDIC може бути використаний для обчислення синуса і косинуса кута θ. Обчислення sin θ та cos θ засноване на обертанні початкового вектора одиничної довжини, який вирівняний по абсцисі ($x_0 = 1$, $y_0 = 0$). Вхідні значення для n ітерацій:

$$x_0 = 1,\ y_0 = 0,\ z_0 = \theta$$

Поклавши початкову умову в рівняннях (2.37), (2.38) для отримання cos θ та sin θ, отримаємо:

$$\frac{x_n}{K_n} = \cos \phi_i$$

$$\frac{y_n}{K_n} = \sin \phi_i$$

Поклавши x0 = $1/K_n$, обертання створює нерозмірний синус і косинус аргументу кута, z_0. Дуже часто значення синуса і косинуса модулюють значення величини. Для використання інших методик (наприклад, таблиці пошуку) потрібна пара множників для отримання модуляції. Метод CORDIC виконує множення як частину операції обертання, і тому виключає необхідність у парі явних множників. На виході CORDIC обертач масштабується підсиленням обертання. Якщо коефіцієнт підсилення неприйнятний, однократне множення на зворотну константу посилення, поставлену перед ротатором CORDIC, дасть незареєстровані результати. Варто зазначити, що складність обертача CORDIC приблизно еквівалентна складності одного множника з однаковим розміром слова.

2.3.Формат чисел з плаваючою комою

Термін «плаваюча кома» походить від того, що до та після десяткової коми немає фіксованої кількості цифр, тобто десяткова кома може фіксуватися. Взагалі, подання з плаваючою комою більш повільне і менш точне, ніж представлення з фіксованою комою, але вони можуть обробляти більший діапазон чисел. Числа з плаваючою комою - це числа, які можуть містити дробову частину. Для прикладу, наступні числа - це числа «плаваючої коми»: 3.0, - 111.5, 1/2, 3E-5 і т. д. Практично в кожній мові є тип даних про температуру відпуску; комп’ютери від ПК до суперкомп'ютерів мають прискорювачі відключення; більшість компіляторів будуть закликані час від часу складати алгоритми з точки зору оцінювання; і практично кожна операційна система повинна реагувати на винятки з точки зору, такі як переповнення.

Перевага представлення з плаваючою комою від подання з фіксованою комою (і цілим числом) полягає в тому, що воно може підтримувати набагато ширший діапазон значень. Наприклад, подання з фіксованою точкою, що має сім десяткових цифр, десяткова крапка вважається розміщеною після п’ятої цифри, може представляти числа 12345.67, 8765.43, 123.00 і так далі, тоді як репрезентація (наприклад, Формат IEEE 754 decimal32) із семи десятковими цифрами може додатково представляти 1,2234567, 123456,7, 0,00001234567, 1234567000000000 тощо. Формат fl крапки вимагає дещо більше місця (для кодування положення коми), тому, зберігаючись у тому самому просторі, числа з плаваючою комою досягають свого більшого діапазону за рахунок трохи меншої точності. Протягом багатьох років у комп’ютерах було використано декілька різних представлень з плаваючою комою, однак протягом останніх десяти років найбільш часто зустрічається представлення, яке визначається стандартом IEEE для арифметики з плаваючою комою (IEEE 754). Це технічний стандарт, встановлений Інститутом інженерів електротехніки та електроніки (IEEE), і найбільш широко використовуваний стандарт для обчислення точки з точки зору повітря, який супроводжується

багатьма апаратними та програмними реалізаціями. Одиничне представлення точності займає 32 біти: знаковий біт, 8 біт для експоненти і 23 для мантиси. Представлення з плаваючою комою, зокрема стандартний формат IEEE, є, безумовно, найпоширенішим способом подання наближення до реальних чисел у комп’ютерах, оскільки воно ефективно обробляється в більшості великих комп'ютерних процесорів.

2.3.1. Нормалізація

Перед тим, як двійкове число з плаваючою комою може бути правильно збережене, його мантиса повинна бути нормалізована. Процес в основному такий же, як і при нормалізації десяткових чисел.

- *Знак*

Ознака двійкового числа, що зазначає точку відведення, представлена одним бітом. 1 біт вказує на негативне число, а 0 біт - на додатне число.

- *Мантиса*

Використовуючи $-3{,}154 \times 10^5$ як приклад, знак від'ємний, мантиса - 3,154, а показник - 5. Дробова частина мантиси - це сума кожної цифри, помножена на степінь 10:

$$.154 = 1/10 + 5/100 + 4/1000$$

У цифрі +11.1011 x 23 знак позитивний, мантиса - 11.1011, а показник - 3. Дробова частина мантиси - це сума послідовних сил 2. У нашому прикладі вона виражається як:

$$.1011 = 1/2 + 0/4 + 1/8 + 1/16$$

- *Експонента*

Короткі дійсні експоненти IEEE зберігаються у вигляді 8-бітних незнакових цілих чисел із зміщенням 127. Давайте використаємо число 1,101 x 25 як приклад. Експонент (5) додається до 127, а сума (132) - двійковою 10100010. Ось кілька прикладів експонентів, спочатку показаних у десятковій формі, потім відрегульованих та остаточних у незнакових двійкових.

Двійковий показник не підписаний, і тому має негативний характер. Найбільший можливий показник – 128 при додаванні до 127, він утворює

255, найбільше незнакове значення, представлене 8 бітами. Орієнтовний діапазон становить від 1,0 x 2^{-127} до 1,0 x 2^{+128}.

Перед тим, як двійкове число, яке зберігається в точці, може бути правильно збережене, його мантису необхідно нормалізувати. Процес в основному такий самий, як і при нормалізації десяткових цифр, що оцінюються. Наприклад, десяткове значення 1234.567 нормалізується як 1.234567 x 103 шляхом переміщення десяткової коми так, що перед десятковою відображається лише одна цифра. Експонент виражає кількість позицій, на які десяткова крапка була переміщена вліво (позитивний показник) або переміщена вправо (від'ємний показник). Так само, двійкове значення 1101.101 з плаваючою комою нормалізується як 1.101101 x 23 шляхом переміщення десяткової коми на 3 позиції вліво , і помноживши на 23.

У нормалізованій мантисі цифра 1 завжди з’являється зліва від десяткової крапки. Фактично, ведучий 1 опущений з мантиси у форматі зберігання IEEE, оскільки він є зайвим.

Знак, експонента і нормалізована мантиса об'єднуються в бінарне коротке реальне подання IEEE. Значення 1,101 x 20 зберігається як знак = 0 (позитивне), мантиса = 101, а експонент = 01111111 (значення показника додається до 127). Значення "1" зліва від десяткової коми випадає з мантиси.

2.3.2. Винятки

Стандарт IEEE визначає п’ять типів винятків, які повинні бути сигналізовані через одно-бітовий флаг статусу у разі їх виникнення.

1) Недійсна операція

Деякі арифметичні операції є недійсними, наприклад, ділення на нульовий або квадратний корінь на від’ємне число. Результатом недійсної операції є NaN (Не число). Існує два типи NaN, тихий NaN (QNaN) і сигнальний NaN (SNaN). Вони мають такий формат, де s - біт знака:

QNaN = s 11111111 10000000000000000000000

SNaN = s 11111111 00000000000000000000001

Результатом кожної недійсної операції є рядок NaN за винятком QNaN або SNaN. Рядок SNaN ніколи не може бути результатом будь-якої операції, може бути сигналізований лише виняток SNaN, і це відбувається всякий раз, коли одним з вхідних операндів є рядок SNaN, інакше буде повідомлено виняток QNaN. Виняток SNaN може, наприклад, використовуватися для сигналізації операцій з неініціалізованими операндами, якщо ми встановимо неініціалізовані операнди SNaN. За умови обробки винятків за замовчуванням, будь-яка операція, що сигналізує про недійсний виняток із операції, і для якої повинен бути наданий результат «точка відключення», забезпечує спокійний NaN. Сигналізація NaN повинні бути зарезервованими операндами, які за обробкою винятків за замовчуванням подають сигнал про недійсний виняток операції для кожної загально-обчислювальної та сигнально-обчислювальної операції.

Нижче наведено деякі арифметичні операції, які є недійсними операціями, і в результаті дають рядок QNaN, і це є сигналом винятку QNaN:

- Будь-яка операція над NaN;
- Додавання або віднімання: ∞ + (-∞);
- Множення: $\pm 0 \times \pm \infty$;
- Ділення: $\pm 0 / \pm 0$ або $\pm \infty / \pm \infty$;
- Квадратний корінь: якщо операнд менше нуля.

2) Ділення на нуль

У математиці ділення називається діленням на нуль, якщо дільник дорівнює нулю. Такий поділ може бути формально виражений як / 0, де а - дивіденд. Чи може це вираз присвоїти чітко визначене значення, залежить від математичної установки. У звичайній арифметиці (реальна кількість) вираз не має значення. У комп'ютерному програмуванні цілочисельний поділ на нуль може призвести до припинення програми або, як у випадку з числами котла, може призвести до спеціального значення, яке не має числа. Поділ будь-якого числа на нуль, окрім нуля, дає результат як результат. Додавання

або множення двох чисел також може призвести до нескінченності. Тож для розмежування двох випадків було застосовано виняток «ділення на нуль».

3) Недоповнення/переповнення

Дві події спричинюють щоб виняток недоповнення був сигналізований, тонкість і втрата точності. Тонкість виявляється після або перед округленням, коли результат лежить між ±2Emin. Втрата точності виявляється, коли результат просто неточний або лише тоді, коли відбувається втрата перенормалізації. У реалізатора є можливість вибрати спосіб виявлення цих подій. Вони повинні бути однаковими для всіх операцій. Впроваджене ядро FPU сигналізує про виняток недоповнення, коли виявляється тонкість після округлення, і в той же час результат є неточним.

Виняток переповнення сигналізується, коли результат перевищує максимальне значення, яке може бути представлене через обмежений діапазон експоненти. Він не сигналізується, коли один з операндів знаходиться у вільності, тому що арифметика у якості дійсності завжди точна. Ділення на нуль також не викликає цього винятку.

Недоповнення/переповнення означає, що експонента результату є занадто великою / малою, щоб бути представленою у полі експоненти. Експонента результату повинна бути розміром 8 біт і повинна бути від 1 до 254, інакше значення не є нормалізованим. Під час додавання двох показників або під час нормалізації може виникнути перевищення. Перевищення за рахунок додавання експонентів може компенсуватися під час віднімання зміщення, що призводить до нормального вихідного значення (нормальна робота). Під час віднімання зміщення для утворення проміжного показника може виникнути недостатність. Якщо проміжний показник <0, то це недолік, який ніколи не можна компенсувати, якщо проміжний показник = 0, то це недолік, який може бути компенсований під час нормалізації, додавши до нього 1.

При переповненні сигнал сигналу переповнення стає високим, і результат переходить у ±∞ (знак, визначений відповідно до знаку входів

множника з плаваючою комою). Коли виникає недостатність, сигнал низького рівня активізації стає високим, і результат переходить до ±0 (знак, визначений відповідно до знаку входів множника з плаваючою комою). Денормалізовані числа сигналізуються до нуля з відповідним знаком, обчисленим з входів, і піднімається сигнал недоповнення. Припустимо, що E1 і E2 є експонентами двох чисел A і B відповідно; показник результату обчислюється:

$$Eresult = E1 + E2 - 127$$

E1 і E2 можуть мати значення від 1 до 254, в результаті чого Eresult має значення від - 125 (2-127) до 381 (508-127), але для нормалізованих чисел Eresult може мати значення лише від 1 до 254.

2.4. Алгоритми перетворення числа з одного формату в інший

Алгоритми з урахуванням діапазону вхідних значень:

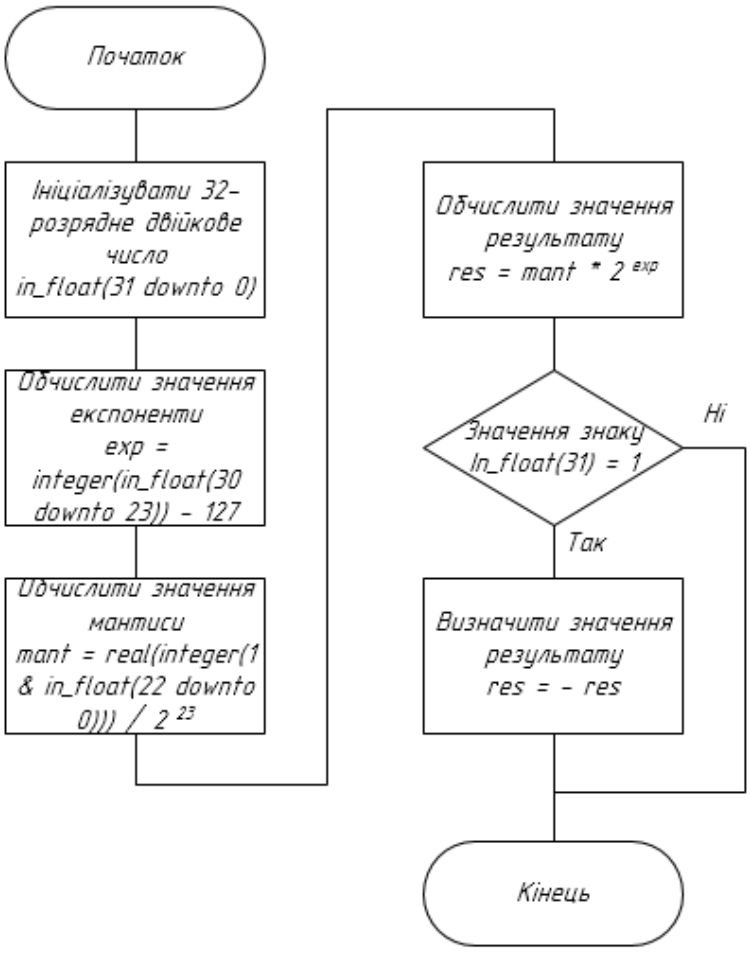


Рис. 2.6. Алгоритм перетворення числа з плаваючою комою у число з фіксованою комою

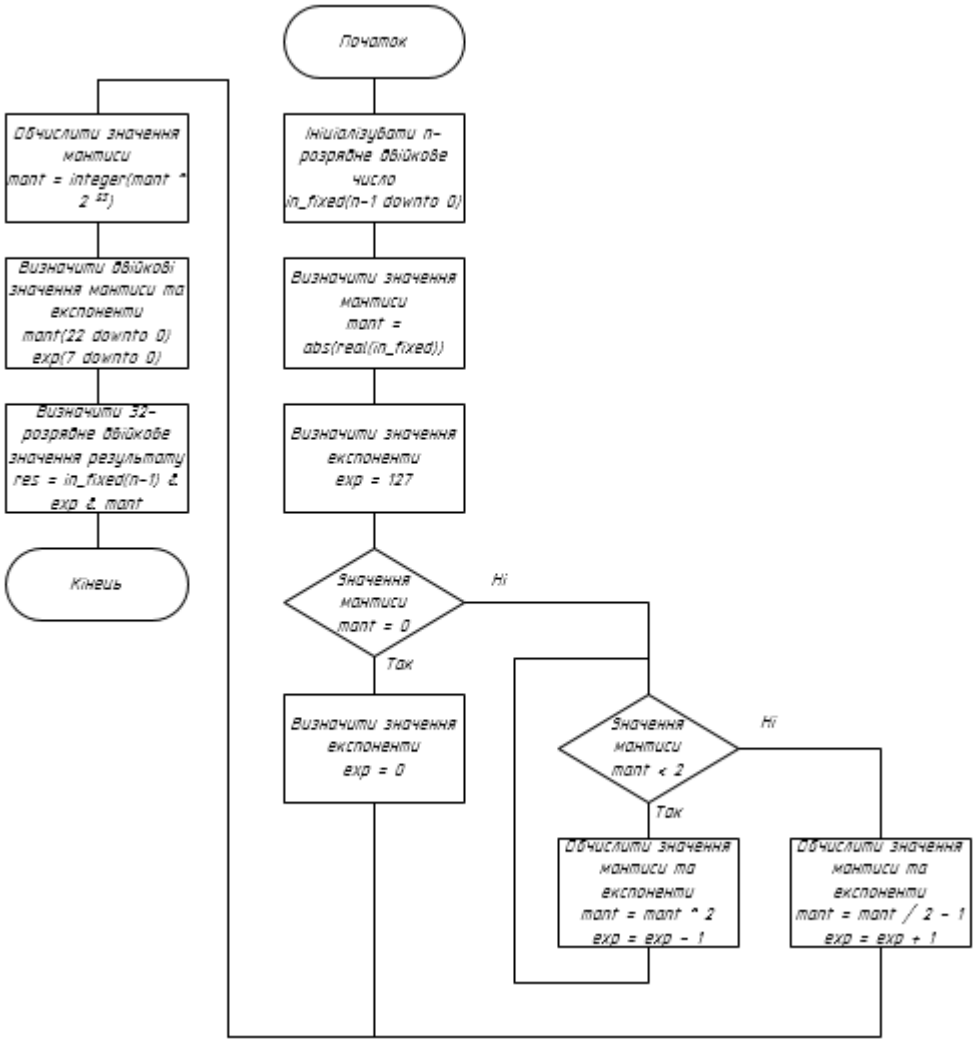


Рис. 2.7. Алгоритм перетворення числа з фіксованою комою у число з плаваючою комою

2.5. Ітераційний та розгорнений процесори CORDIC

Ітераційна архітектура CORDIC може бути отримана просто шляхом дублювання кожного з трьох різницьких рівнянь у апаратному забезпеченні, як показано на рис. 2.8. Функція рішення d_i керується знаком регістра y або z залежно від того, керується вона в режимі обертання або у векторному режимі. В процесі роботи початкові значення завантажуються через мультиплексори в регістри x, y і z. Потім на кожному з наступних n тактових циклів значення з регістрів передаються через перемикачі та суматори-віднімачі, а результати розміщуються назад у регістрах. Зсуви змінюються на кожній ітерації, щоб викликати бажаний зсув для ітерації. Крім того, адреса ПЗУ збільшується на кожній ітерації, так що відповідне значення

елементарного кута подається на z суматор-віднімач. На останній ітерації результати зчитуються безпосередньо з суматорів-віднімачів. Очевидно, що потрібна проста машина стану, яка слідкує за поточною ітерацією та вибирає ступінь зсуву та адресу ROM для кожної ітерації.

Дизайн, зображений на рис. 2.8, використовує загальнодоступні шляхи передачі даних (звані біт-паралельним дизайном). Біт-паралельні перемикачі змінних зсувів не добре відображають архітектури ПЛІС через необхідний великий вентилятор. Якщо вони реалізовані, для цих перемикачів зазвичай потрібно декілька логічних шарів (тобто сигнал повинен пройти через декілька комірок ПЛІС). Результат - повільна конструкція, яка використовує велику кількість логічних комірок.

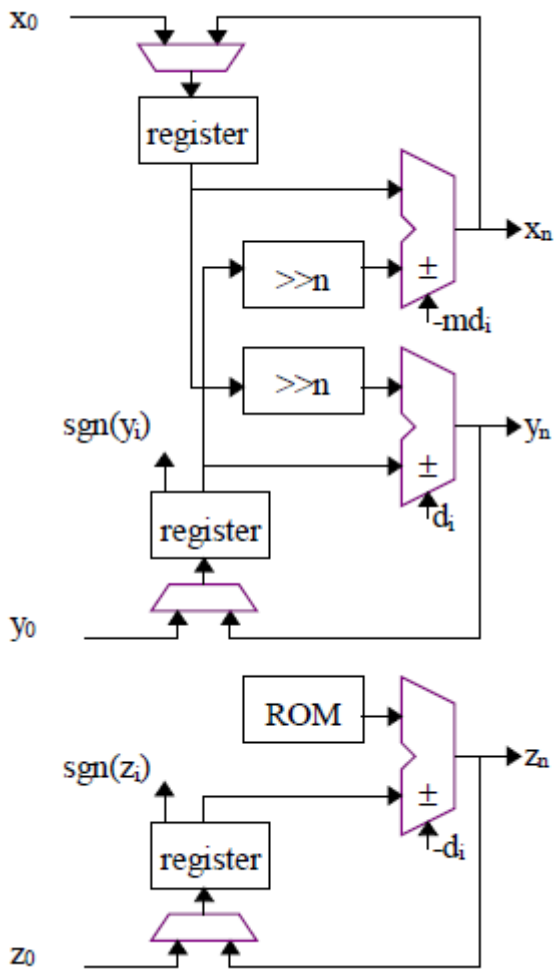


Рис. 2.8. Структура ітераційного CORDIC

Значно більш компактна конструкція можлива з використанням бітової послідовної арифметики. Спрощений взаємозв'язок і логіка в бітовій серійній конструкції дозволяє йому працювати зі значно більшою тактовою частотою, ніж еквівалентна бітова паралельна конструкція. Зрозуміло, дизайн також повинен мати тактовий час для кожної ітерації (w - ширина слова даних). Бітова послідовна конструкція складається з трьох бітових послідовних суматорів-віднімачів, трьох регістрів зсуву та послідовної пам'яті лише для читання (ROM). Кожен регістр змін має довжину, рівну ширині слова. Існує також декілька решіток або мультиплексорів для вибору відводів регістрів зсуву для правильних зміщених поперечних термінів (зсув здійснюється за допомогою затримки бітів у бітових послідовних системах). Бітова послідовна архітектура CORDIC показана на рис. 2.9. У цій конструкції потрібні w тактів для кожної з n ітерацій, де w - точність суматорів. Під час роботи мультиплексори навантаження зліва відкриваються на w тактові періоди, щоб ініціалізувати регістри x, y та z (ці регістри також можуть бути паралельно завантажені для ініціалізації). Після завантаження дані переміщуються праворуч через послідовне додавання-віднімання та повертаються в лівий кінець регістра. Кожна ітерація вимагає w годин, щоб повернути результат у регістр. На початку кожної ітерації машина управління керує ознакою регістру y (або z) і відповідно встановлює елементи додавання / віднімання. Відповідне відключення регістра для перехресних умов також вибирається на початку кожної ітерації. Під час n-ї ітерації результати можна читати з виходів послідовних суматорів, тоді як наступні дані ініціалізації зміщуються в регістри.

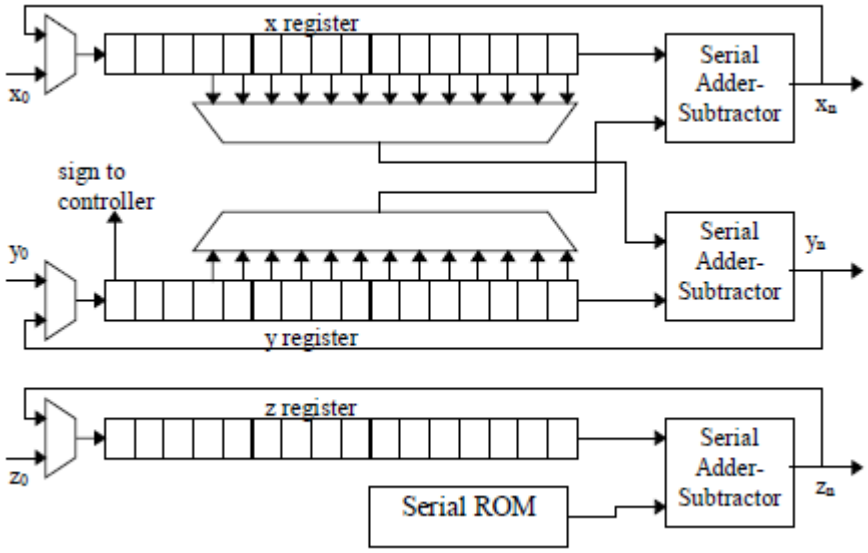


Рис. 2.9. Біт-серійний ітераційний CORDIC

Простота бітової послідовної конструкції видно з рис. 2.9. Навіть у цьому випадку проводка мультиплексорів зсувного перемикача може створити проблеми в деяких ПЛІС (це одне місце, де можуть стати в нагоді тришарові довгі лінії). Тим не менш, взаємозв'язок мінімальний, а логіка між регістрами проста. Ця комбінація дозволяє розряджати бітові тактові частоти поблизу максимальної частоти перемикання ПЛІС. Можливість використання екстремальних частот тактових частот обумовлює велику кількість тактових циклів, необхідних для завершення кожного обертання.

Розглянутий до цього процесор CORDIC є ітераційним, що означає, що процесор повинен виконувати ітерації в n-кратній швидкості передачі даних. Процес ітерації може розгорнутися [17], так що кожен з n елементів обробки завжди виконує однакову ітерацію. Розгорнений або трубкоподібний процесор CORDIC показаний на рис. 2.10. Розгорнення процесора призводить до двох значних спрощень. По-перше, перемикачі - це фіксований зсув, це означає, що вони можуть бути реалізовані в електропроводці. По-друге, значення пошуку для кутового акумулятора розподіляються як константи кожному суматору в ланцюзі кутового акумулятора. Ці константи можуть бути провідними, а не вимагати місця для

зберігання. Весь процесор CORDIC зводиться до масиву взаємопов'язаних суматорів-віднімачів. Потреба в регістрах також усувається, що робить розгорнений процесор суворо комбінаторним. Затримка через результуючу схему була б істотною, але час обробки скорочується від часу, необхідного ітераційній схемі (якщо не чим іншим, як часом встановлення та утримування регістра). У більшості випадків, особливо в ПЛІС, не має сенсу використовувати такий великий комбінаторний контур. Розгорнений процесор легко конвеєризується, вставляючи регістри між суматорами відніманнями. У випадку більшості архітектур ПЛІС вже є регістри, присутні у кожній логічній комірці, тому додавання регістрів конвеєра не приносить витрат на обладнання.

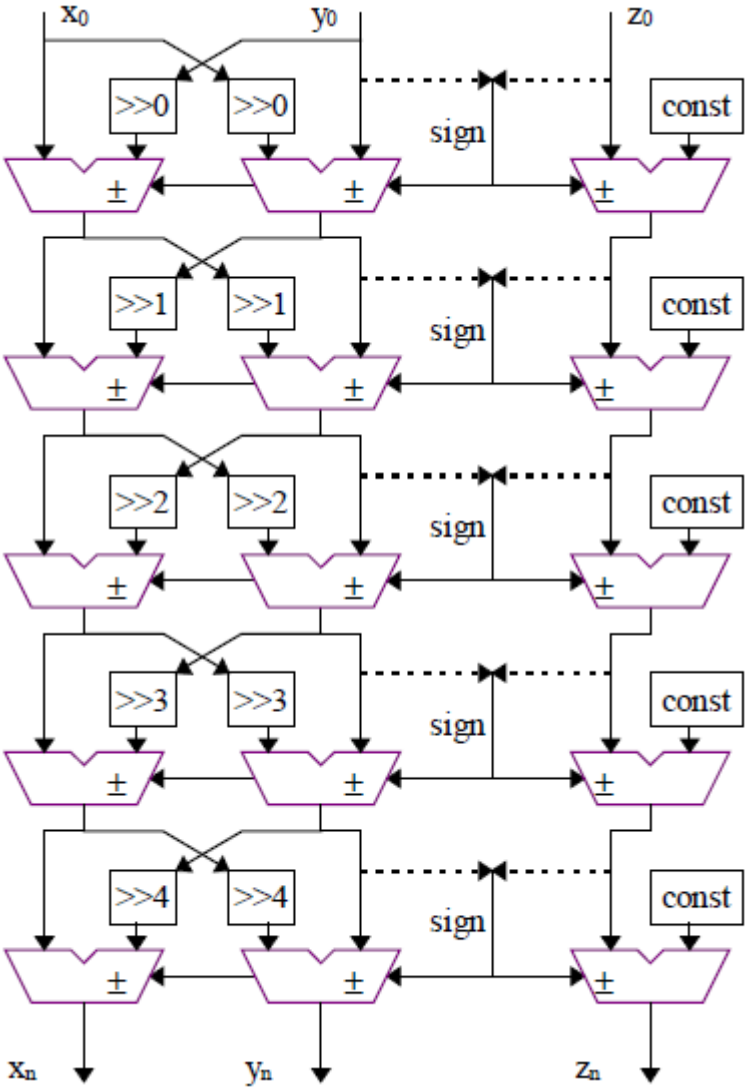


Рис. 2.10. Трубкоподібний процесор CORDIC

Трубкоподібний процесор також може бути перетворений на біт-серійну конструкцію. Кожен суматор-віднімач замінюється послідовним суматором-віднімачем, розділеним w-регістрами зсуву. Регістри зсуву необхідні для отримання знаку елемента y або z до того, як перші біти (1sbs) дістануться наступного суматора-віднімача. Праворуч зміщені перехрестя взяті з фіксованих відводів у регістрах зсуву. Також потрібен певний метод розширення знаків для зміщених суматорів.

Більш висока продуктивність вимагає або декількох розрядних послідовних процесорів, що працюють паралельно, або трубкоподібного паралельного конвеєра. До недавнього часу ПЛІС не мали необхідного поєднання логіки та ресурсу маршрутизації для створення паралельного процесора. Цей бар'єр здебільшого пояснюється великою кількістю перехресних маршрутів, необхідних між регістрами x та y на кожній стадії трубки. Крім того, продуктивність зменшується, оскільки ширина слова збільшується через тривалість поширення переносу через суматори.

ВИСНОВКИ ДО РОЗДІЛУ 2

1. Після аналізу алгоритму CORDIC та видів процесорів даного алгоритму було вирішено використати трубкоподібний CORDIC, як основу для модуля з конвеєризованим методом обробки вхідних даних.
2. Було розроблено алгоритм для перетворення вхідних даних формату з плаваючою комою у дані формату з фіксованою комою для подальші їх обробки в CORDIC та алгоритм навпаки для отримання вихідних даних потрібного формату.

РОЗДІЛ 3
ПРОЕКТУВАННЯ МОДУЛЯ

3.1. Опис модуля мовою VHDL

Опис конвеєризованого CORDIC:

```
x_array <= X & x_pl;
y_array <= Y & y_pl;
z_array <= Z & z_pl;

cordic: process(Clock, Reset) is
  variable negative : boolean;
begin
  if Reset = RESET_ACTIVE_LEVEL then
    Busy <= '1';
    cur_cycle <= 1;
    cur_iter <= 1;
    x_pl <= (others => (others => '0'));
    y_pl <= (others => (others => '0'));
    z_pl <= (others => (others => '0'));

  elsif rising_edge(Clock) then
    if cur_iter /= ITERATIONS+1 then
      negative := z_array(cur_iter-1)(z'high) = '1';
      if negative then
        x_pl(cur_iter) <= x_array(cur_iter-1) + (y_array(cur_iter-1)
          / 2**(cur_iter-1));
        y_pl(cur_iter) <= y_array(cur_iter-1) - (x_array(cur_iter-1)
          / 2**(cur_iter-1));
        z_pl(cur_iter) <= z_array(cur_iter-1) + ATAN_TABLE(cur_iter-1);
      else
        x_pl(cur_iter) <= x_array(cur_iter-1) - (y_array(cur_iter-1)
          / 2**(cur_iter-1));
        y_pl(cur_iter) <= y_array(cur_iter-1) + (x_array(cur_iter-1)
          / 2**(cur_iter-1));
        z_pl(cur_iter) <= z_array(cur_iter-1) - ATAN_TABLE(cur_iter-1);
      end if;
      cur_iter <= cur_iter + 1;
    elsif cur_cycle = 2 then
         Busy <= '0';
         cur_iter <= ITERATIONS+1;
    else
         cur_cycle <= cur_cycle + 1;
         cur_iter <= 1;
    end if;
  end if;
end process;

X_result <= x_array(x_array'high);
Y_result <= y_array(y_array'high);
Z_result <= z_array(z_array'high);
```

Рис. 3.1. Опис конвеєризованого CORDIC

Функціональна схема конвеєризованого CORDIC:

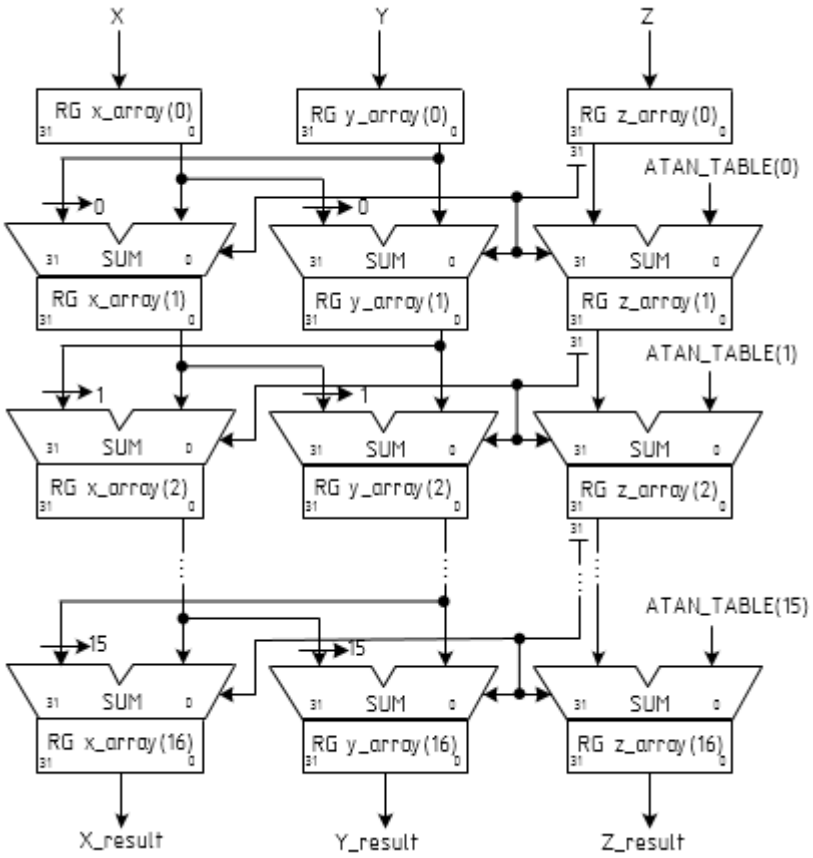


Рис. 3.2. Функціональна схема конвеєризованого CORDIC

Опис функцій перетворення чисел у форматі з плаваючою комою в числа у форматі з фіксованою комою та навпаки:

```
function float_to_signed(float_in: std_logic_vector(31 downto 0);
      amplitude : real) return signed is
  variable exp, mant : integer;
  variable mant_real, res_real : real;
  variable res : signed(31 downto 0);
begin
    exp := to_integer(unsigned(float_in(30 downto 23))) - 127;
    mant := to_integer(unsigned('1' & float_in(22 downto 0)));
    mant_real := real(mant) / (2.0**23);
    res_real := mant_real * 2.0**exp;
    if float_in(31) = '1' then
        res_real := - res_real;
    end if;
    res := to_signed(integer((res_real / amplitude)
      * (2.0**31 - 1.0)), 32);
    return res;
end function;
```

Рис. 3.3. Функція перетворення числа з плаваючою комою у число з фіксованою комою

```
function signed_to_float(signed_in : signed(31 downto 0);
      amplitude : real) return std_logic_vector is
  variable res : std_logic_vector(31 downto 0);
  variable sign : std_logic;
  variable exp : std_logic_vector(7 downto 0);
  variable mant : std_logic_vector(22 downto 0);
  variable in_abs: signed(31 downto 0);
  variable in_int, exp_int, mant_int : integer;
  variable mant_real : real;
begin
    sign := signed_in(31);
    in_abs := abs(signed_in);
    in_int := to_integer(unsigned(in_abs(30 downto 0)));
    mant_real := real(in_int) * amplitude / (2.0**31 - 1.0);
    exp_int := 127;
    if in_int /= 0 then
        while mant_real < 2.0 loop
          mant_real := mant_real * 2.0;
          exp_int := exp_int - 1;
        end loop;
        mant_real := mant_real / 2.0;
        exp_int := exp_int + 1;
        mant_int := integer((mant_real - 1.0) * (2.0**23));
    else exp_int := 0; mant_int := 0; sign := '0';
    end if;
    exp := std_logic_vector(to_unsigned(exp_int, 8));
    mant := std_logic_vector(to_unsigned(mant_int, 23));
    res := sign & exp & mant; return res;
end function;
```

Рис. 3.4. Функція перетворення числа з фіксованою комою у число з плаваючою комою

Опис сутності модуля мовою програмування VHDL виглядає наступним чином:

```
entity sincos_pipelined is
  generic (
    MAGNITUDE  : real := 1.0;
    SIZE       : positive := 32;
    ITERATIONS : positive := 16;
    RESET_ACTIVE_LEVEL : std_ulogic := '1'
  );
  port (
    Sin   : out signed(SIZE-1 downto 0);
    Cos   : out signed(SIZE-1 downto 0)
  );
end entity;
```

Рис. 3.5. Сутність модуля

Даний опис містить generic зону налаштування, в якій встановлюється завчасно необхідна розрядність вхідних та вихідних даних, кількість ітерацій алгоритму, а також необхідні константи. Серед переліку портів пристрою всього 2 порти виходу для отриманих результатів. Всі порти налаштовані на роботу з типом STD_LOGIC зі стандартної бібліотеки VHDL IEEE.STD_LOGIC_1164, а отже, напряму чи через функції перетворення типів можуть бути поєднані з широким спектром можливих сигналів системи, в яку буде вбудовано модуль.

Модуль містить імпортовані компоненти реалізованого CORDIC та керуючого автомату CLK_GEN, що описано наступним чином:

```
component clk_gen is
  generic(
      clk_period : time := 10ns;
      angle_gain_coef : real := 200.0;
      SIZE       : positive := 32;
      ITERATIONS : positive := 16;
      RESET_ACTIVE_LEVEL : std_ulogic := '1'
      );
  port(
      CLK : out STD_ULOGIC;
      RST : out STD_ULOGIC;
      Angle : out std_logic_vector(SIZE-1 downto 0)
      );
  end component;
```

Рис. 3.6. Компонента керуючого автомату

```
component cordic_pipelined is
  generic (
      SIZE               : positive;
      ITERATIONS         : positive;
      RESET_ACTIVE_LEVEL : std_ulogic
  );
  port (
      Clock : in std_ulogic;
      Reset : in std_ulogic;
      Busy  : out std_ulogic;
      X : in signed(SIZE-1 downto 0);
      Y : in signed(SIZE-1 downto 0);
      Z : in signed(SIZE-1 downto 0);
      X_result : out signed(SIZE-1 downto 0);
      Y_result : out signed(SIZE-1 downto 0);
      Z_result : out signed(SIZE-1 downto 0)
  );
end component;
```

Рис. 3.7. Компонента CORDIC

Порти та константи даних імпортованих компонент асоціюються з відповідними сигналами модуля, що описується наступним чином:

```
U1:clk_gen
generic map (
    SIZE => SIZE,
    ITERATIONS => ITERATIONS,
    RESET_ACTIVE_LEVEL => RESET_ACTIVE_LEVEL
) port map (
    CLK => Clock,
    RST => Reset,
    Angle => Angle
);
```

Рис. 3.8. Прив’язання автомату до модуля

```
U2: cordic_pipelined
  generic map (
    SIZE => SIZE,
    ITERATIONS => ITERATIONS,
    RESET_ACTIVE_LEVEL => RESET_ACTIVE_LEVEL
  ) port map (
    Clock => Clock,
    Reset => Reset,
    Busy => Busy,
    X => xa,
    Y => ya,
    Z => za,
    X_result => Cos_loc_new,
    Y_result => Sin_loc_new,
    Z_result => open
  );
```

Рис. 3.9. Прив’язання реалізованого CORDIC до модуля

Структурна схема модуля:

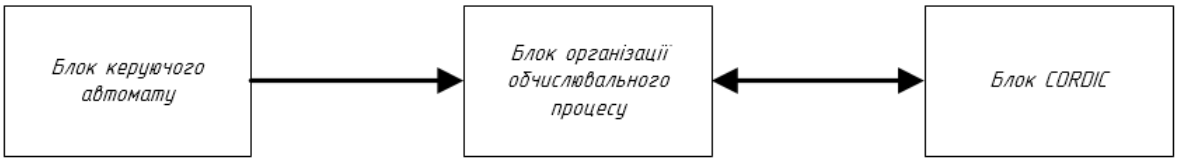


Рис. 3.10. Структурна схема модуля

Операційний блок відповідає за реалізацію обчислювального процесу модуля, а саме отримання потрібного формату вхідних даних для імпортованої компоненти конвеєризованого CORDIC та перетворення отриманих від цієї ж компоненти вихідних даних у потрібний формат. Даний блок описано у вигляді process конструкції та виглядає наступним чином:

```
P: process(Clock, Reset, Busy) is
  constant Y : signed(SIZE-1 downto 0) := (others => '0');
  constant X : signed(SIZE-1 downto 0) :=
    to_signed(integer((MAGNITUDE / cordic_gain(ITERATIONS))
      * (2.0**(SIZE-1) - 1.0)), SIZE);
  variable new_values : boolean;
begin
  if Reset = RESET_ACTIVE_LEVEL then
    xa <= (others => '0');
    ya <= (others => '0');
    za <= (others => '0');
  elsif rising_edge(Clock) then
    Angle_signed <= float_to_signed(Angle, MATH_PI);
    adjust_angle(X, Y, Angle_signed, xa, ya, za);
  end if;

  new_values := Busy = '0';
    if new_values then
          Sin_loc <= Sin_loc_new;
          Cos_loc <= Cos_loc_new;
    end if;
end process;
```

Рис. 3.11. Операційний блок

Опис функцій, які використовує операційний блок:

```
function cordic_gain(iterations : positive) return real is
  variable g : real := 1.0;
begin
  for i in 0 to iterations-1 loop
    g := g * sqrt(1.0 + 2.0**(-2*i));
  end loop;
  return g;
end function;
```

Рис. 3.12. Значення приросту CORDIC

```
procedure adjust_angle(x, y, z : in signed; signal xa, ya, za : out signed) is
  variable quad : unsigned(1 downto 0);
  variable zp : signed(z'length-1 downto 0) := z;
  variable yp : signed(y'length-1 downto 0) := y;
  variable xp : signed(x'length-1 downto 0) := x;
begin
  quad := unsigned(zp(zp'high downto zp'high-1));
  if quad = 1 or quad = 2 then
    xp := -xp;
    yp := -yp;
    zp := (not zp(zp'left)) & zp(zp'left-1 downto 0);
  end if;
  xa <= xp;
  ya <= yp;
  za <= zp;
end procedure;
```

Рис. 3.13. Корекція значення кута

Приклад опису керуючого автомату:

```
process begin
    RST <= '1';
    CLK <= '0';
    wait for clk_period;
    RST <= '0';
    CLK <= '1';
    clk_loc <= '1';
    cur_iter <= 0;
    angle_real <= - MATH_PI;
    loop
        wait for clk_period;
        cur_iter <= cur_iter + 1;
        clk_loc <= not clk_loc;
        CLK <= not clk_loc;
        angle_signed <= to_signed(integer( (angle_real / MATH_PI)
        * (2.0**(SIZE-1) - 1.0) ), SIZE);
        Angle <= signed_to_float(angle_signed, MATH_PI);
        if cur_iter = ITERATIONS * 4 + 4 then
            RST <= '1';
            cur_iter <= 0;
            if integer((angle_real / MATH_PI) * angle_gain_coef)
            < integer(angle_gain_coef) then
                angle_real <= angle_real + (MATH_PI / angle_gain_coef);
            else angle_real <= - MATH_PI;
            end if;
        else RST <= '0';
        end if;
    end loop;
end process;
```

Рис. 3.14. Керуючий автомат

3.2. Моделювання розробленого модуля

Для перевірки правильності роботи модуля та правильності отриманих результатів необхідно врахувати що вхідні та вихідні, а також проміжні дані є векторами бітів у форматі двійкових чисел з плаваючою або фіксованою комою, які лежать в деякому діапазоні значень.

Нехай вхідне значення кута змінюється з кожним тактом на деяке мале значення приросту, при чому знаходячись в допустимому інтервалі вхідних значень. Для наочної перевірки правильності роботи модуля нехай значення на виході представлене у форматі з фіксованою комою без знаку. Маємо такі результати:

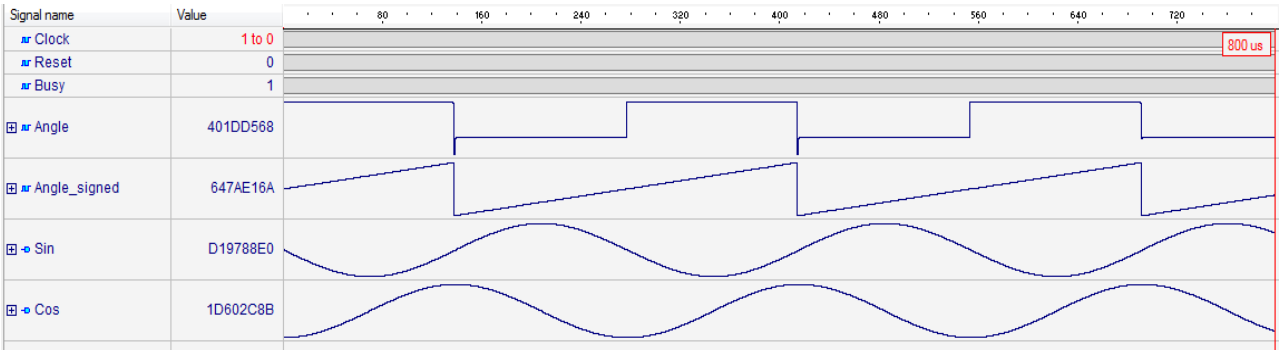


Рис. 3.15. Результати моделювання з числами на виході у форматі з фіксованою комою

Результати моделювання для аналогічного потоку вхідних даних зі значеннями на виході, представленими у форматі з плаваючою комою:

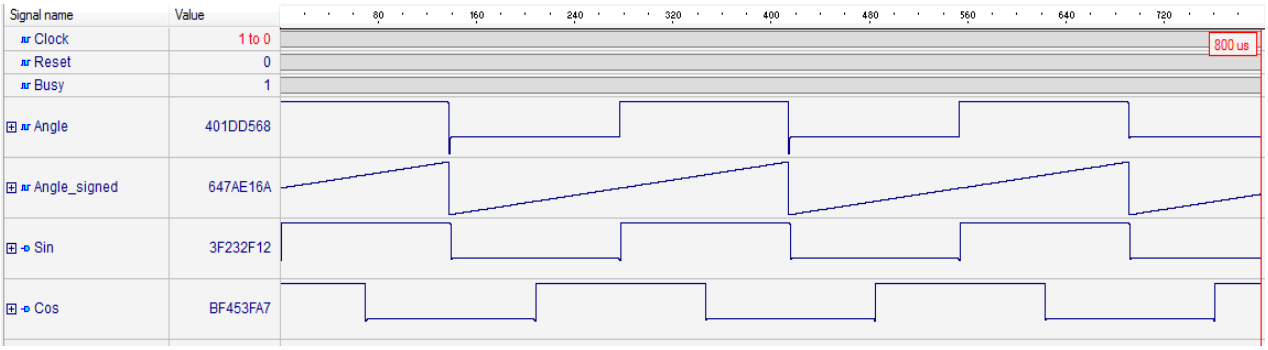


Рис. 3.16. Результати моделювання з числами на виході у форматі з плаваючою комою

ВИСНОВКИ ДО РОЗДІЛУ 3

1. У ході проектування модуля на мові програмування VHDL були описані конвеєризований алгоритм CORDIC, функції перетворення чисел з одного формату представлення в інший та допоміжні функції для організації обчислювального процесу.
2. Було задано приклад керуючого автомату для моделювання пристрою та виконано моделювання для форматів вхідних і вихідних даних за умовою та формату вихідних даних для наочної перевірки результатів.

РОЗДІЛ 4
ТЕСТУВАННЯ МОДУЛЯ

ПЛІС розшифровуються як програмовані логічні інтегральні схеми, які можуть бути налаштовані замовником або дизайнером після виробництва. Програмовані логічні інтегральні схеми називають так, тому що замість того, щоб мати структуру, схожу на PAL або інший програмований девайс, вони структуровані дуже схоже на ASIC масив воріт. Це робить ПЛІС дуже приємними для використання в прототипуванні ASIC або в місцях, де і ASIC з часом буде використовуватися [18]. Для прикладу ПЛІС, можливо, використовується в дизайні, який потребує швидкого виходу на ринок незалежно від вартості. Пізніше ASIC може бути використаний замість ПЛІС, коли обсяг виробництва збільшується, щоб зменшити собівартість. ПЛІС програмуються за допомогою логічної схеми або вихідного коду на мові опису апаратних засобів (HDL), щоб вказати, як буде працювати чіп. ПЛІС містять програмовані логічні компоненти, які називаються "логічними блоками", та ієрархію налаштованих взаємозв'язків, які дозволяють блокам "з'єднуватися". Програмовані логічні блоки називаються блоками, що настроюються, а налаштовані між собою з'єднання називаються комутаційні блоки. Логічні блоки (ЛБ) можуть бути налаштовані для виконання комплексних комбінаційних функцій або просто простих логічних операцій, таких як AND і XOR. У більшості ПЛІС логічні блоки також містять елементи пам'яті, які можуть бути простими оперативними або більш повними блоками пам'яті [19].

4.1. Архітектура ПЛІС

У кожного постачальника ПЛІС є своя архітектура ПЛІС, але загалом вони всі є варіацією тієї, що показана на рис. 3.1. Архітектура складається з налаштованих логічних блоків, налаштованих блоків вводу / виводу та програмованого взаємозв'язку. Також буде створена тактова схема для передачі тактових сигналів до кожного логічного блоку, а також можуть бути

доступні додаткові логічні ресурси, такі як ALU, пам'ять та декодери. Два основні типи програмованих елементів для ПЛІС - це статична оперативна пам’ять та запобіжники. Схема програми повинна бути відображена у FPGA з достатніми ресурсами. Незважаючи на те, що кількість ЛБ та необхідний введення / виведення легко визначається з проектування, кількість необхідних треків може значно відрізнятися навіть серед конструкцій з однаковою логікою. [20]

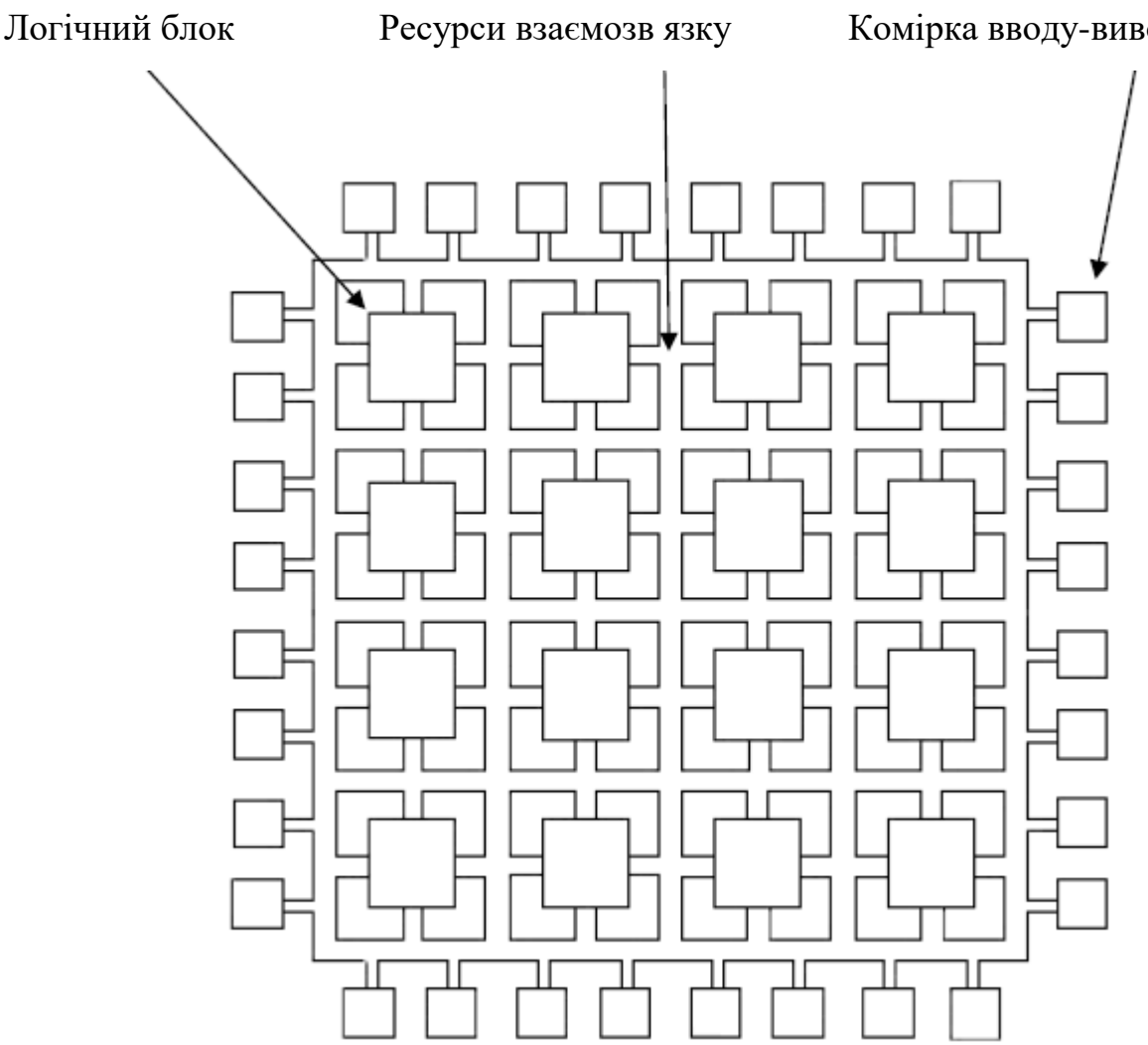


Рис. 4.1. Архітектура ПЛІС

4.1.1. Налаштовувані логічні блоки

Налаштовувані логічні блоки містять логіку для ПЛІС. У великій зерновій архітектурі ці ЛБ містять достатню логіку для створення невеликої

державної машини. У тонкозернистій архітектурі, більше як у справжнього масиву воріт ASIC, ЛБ буде містити лише дуже базову логіку. Діаграма на рис. 3.2 вважатиметься великим зерновим блоком. Він містить ОЗУ для створення довільних комбінаторних логічних функцій. Він також містить додаткові додатки для тактових елементів зберігання та мультиплексори для маршрутизації логіки всередині блоку та до зовнішніх ресурсів. Мультиплексори також дозволяють вибір полярності та скидання та чіткий вибір входу.

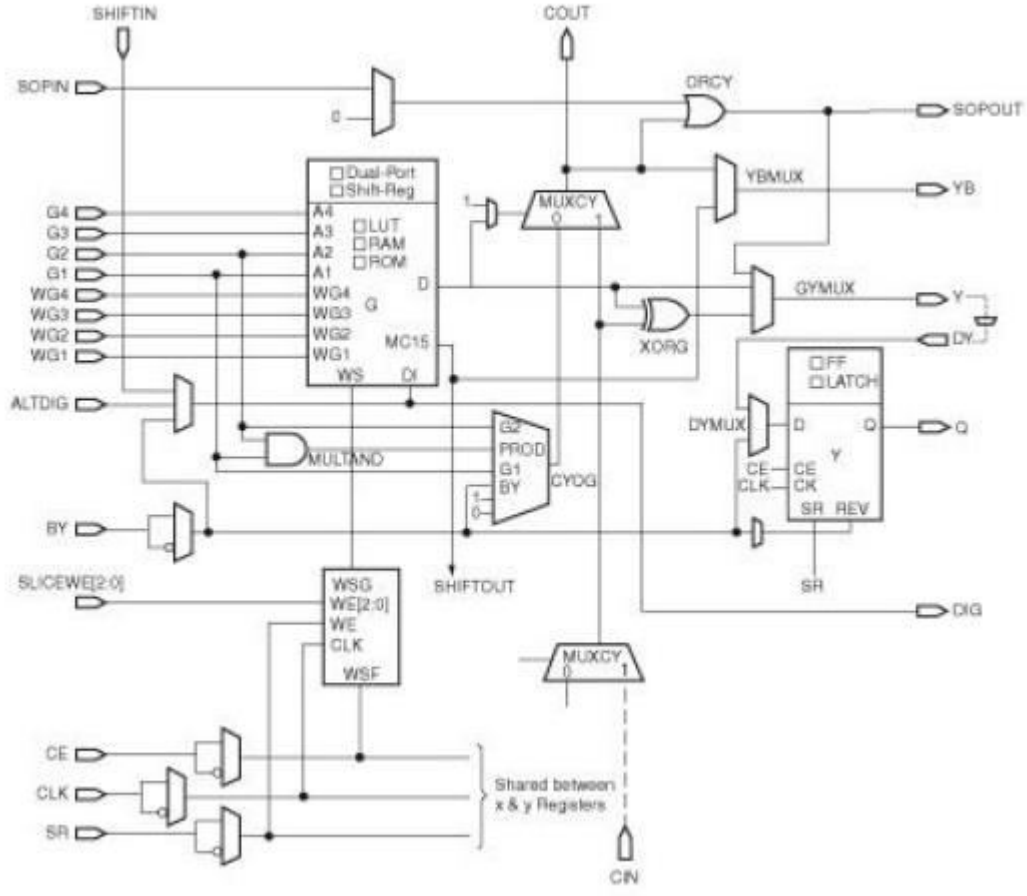


Рис. 3.2. Налаштовуваний логічний блок

4.1.2. Налаштовувані блоки вводу-виводу

Блок блоку вводу-виводу, який можна налаштувати, використовується для подачі сигналів на мікросхему та відсилання їх знову. Він складається з вхідного буфера і вихідного буфера з трьома керуючими виходами стану та відкритого колектора. Зазвичай на виходах є резистори, що підтягуються, а

іноді і резистори, що тягнуться. Полярність виходу, як правило, може бути запрограмована на активний високий або активний низький вихід, і часто скорочення виходу може бути запрограмовано на швидкий або повільний час підйому і падіння. Крім того, часто є додаткові виходи на виходах, так що синхронізовані сигнали можуть виводитися безпосередньо на штифти, не стикаючись із значною затримкою. Це робиться для входів, щоб не було великої затримки сигналу до досягнення додаткового сигналу, який би збільшив потребу в пристрої.

4.1.3. Програмований взаємозв’язок

Взаємозв'язок ПЛІС сильно відрізняється, ніж у CPLD, але досить схожий на зв'язок масиву воріт ASIC. На рис. 3.3 видно ієрархію взаємопов’язаних ресурсів. Існують довгі лінії, які можна використовувати для підключення критичних ЛБ, які фізично далекі один від одного на мікросхемі, не викликаючи великих затримок. Їх також можна використовувати як автобуси в межах мікросхеми. Існують також короткі лінії, які використовуються для з'єднання окремих ЦЛБ, розташованих фізично близько один від одного. Часто існує одна або кілька матриць комутації, як у CPLD, щоб з'єднати ці довгі та короткі лінії разом певними способами. Програмовані вимикачі всередині мікросхеми дозволяють підключити ЛБ для з'єднання ліній та з'єднань ліній між собою та до матриці комутаторів. Три державні буфери використовуються для підключення багатьох ЛБ до довгої лінії, створюючи шину. Спеціальні довгі лінії, які називаються глобальними годинниковими лініями, спеціально розроблені для низького опору і, отже, швидкого часу поширення. Вони підключені до буферів тактових годин та до кожного тактового елемента у кожному ЛБ. Ось так розподіляються годинники по всій ПЛІС.

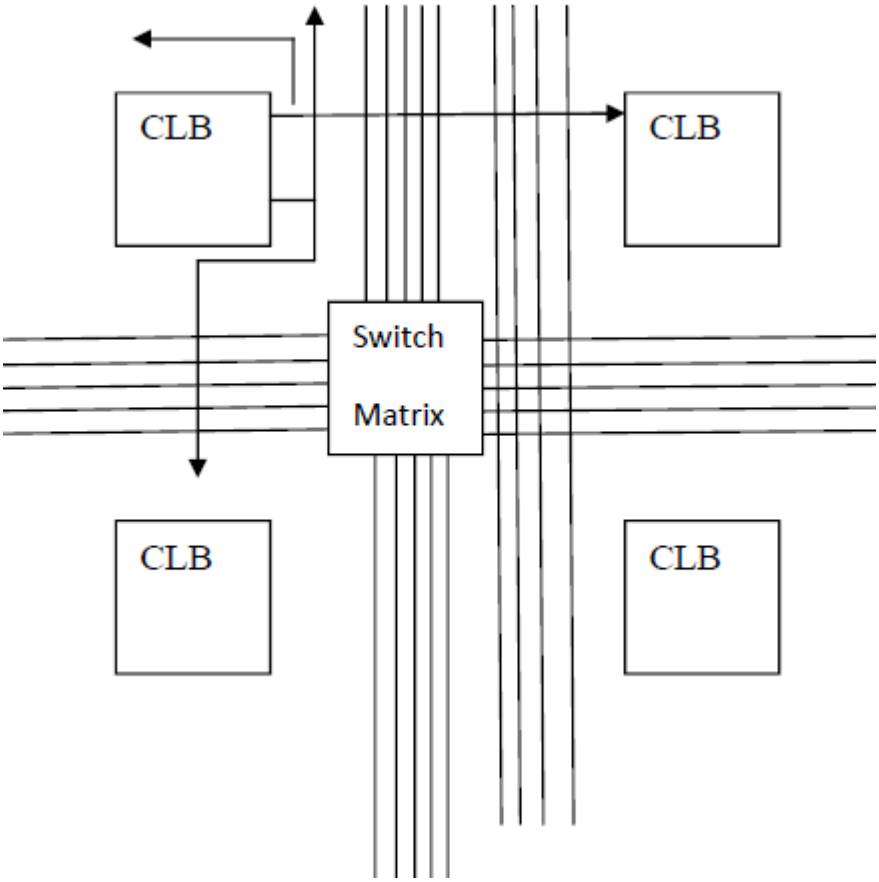


Рис. 3.3. Програмований взаємозв язок в ПЛІС

4.1.4. Годинникова схема

Спеціальні блоки вводу-виводу зі спеціальними буферами тактових частот, відомі як драйвери годин, розподіляються навколо мікросхеми. Ці буфери з'єднані з вхідними колодками тактових сигналів та передають сигнали тактових сигналів на глобальні тактові лінії, описані вище. Ці годинникові лінії розраховані на низький час перекосу та швидкий час розповсюдження. Синхронна конструкція є обов'язковою для ПЛІС, оскільки не можна гарантувати абсолютного перекосу та затримки [21]. Тільки при використанні тактових сигналів з тактових буферів можна гарантувати відносні затримки та час перекосу.

4.1.5. Можливості проектування

У цьому розділі розглядаються можливості проектування за допомогою FPGA. Це весь процес проектування пристрою, який гарантує, що ви не

будете випускати жодних кроків і матимете найкращі шанси отримати назад робочий прототип, який правильно функціонує у вашій системі [22]. Конструкція складається з етапів:

- Декларація проекту в HDL
- Перевірка функціональності поведінки
- Синтез
- Трансляція
- Мапінг
- PAR-процес
- Програмування ПЛІС

1) Суб'єкт дизайну

На цьому кроці розроблена базова архітектура системи, кодована мовою опису апаратних засобів, як VHDL або Verilog.

2) Моделювання поведінки

Після фази проектування код перевіряється за допомогою програмного моделювання, тобто ModelSim або XilinX ISE Simulator для різних входів, щоб генерувати виходи, і якщо він перевіряється, то ми продовжимо подальші зміни в іншому випадку, і необхідне виправлення буде зроблено в коді HDL. Це називається моделюванням поведінки. Моделювання - це постійний процес, поки розробляється дизайн. Невеликі ділянки конструкції слід імітувати окремо, перш ніж підключити їх до більших секцій. Для отримання правильної функціональності буде багато ітерацій проектування та моделювання [23]. Після закінчення проектування та моделювання необхідно провести ще один перегляд дизайну, щоб можна було перевірити дизайн. Важливо змусити інших переглядати симуляції та переконатися, що нічого не пропущено і що не було зроблено неправильного припущення. Це одне з найважливіших переглядів, тому що лише за допомогою правильного та повного моделювання ви дізнаєтесь, що ваш чіп буде правильно працювати у вашій системі.

3) Синтез дизайну

Після правильних результатів моделювання конструкція потім синтезується. Під час синтезу інструмент Xilinx ISE робить наступні операції:

- Компіляція HDL: Інструмент збирає всі підмодулі в головному модулі, якщо виникають проблеми з модулем, тоді перевірте синтаксис коду, написаного для проектування.
- Аналіз ієрархії проектування: аналіз ієрархії дизайну.
- Синтез HDL: синтезує код HDL з точки зору таких компонентів, як мультиплексори, суматори/віднімачі, лічильники, регістри, компаратори, декодери тощо.
- Розширений синтез HDL: У синтезі низького рівня блоки, що синтезуються в синтезі HDL та розширений синтез HDL, далі визначаються з точки зору блоків низького рівня, таких як буфери, шукайте таблиці. Він також оптимізує логічні об'єкти в дизайні, усуваючи зайву логіку, якщо така є. Потім інструмент генерує файл netlist (файл NGC), а потім оптимізує його. Кінцевий вихідний файл netlist має розширення .ngc. Цей файл NGC містить як проектні дані, так і обмеження. Мета оптимізації може бути попередньо визначена як швидша швидкість роботи або мінімальна площа реалізації перед запуском цього процесу. Намагання оптимізації рівня вимагає збільшення часу процесора (тобто часу на розробку), оскільки кілька алгоритмів оптимізації намагаються отримати найкращий результат для цільової архітектури. [24]

  4) Реалізація дизайну

  Процес реалізації дизайну складається з наступних підпроцесів:

- Трансляція: Процес перекладу об'єднує всі вхідні нет-листи та проектні обмеження виводів Xilinx NGD (Рідна інформація та загальна база даних). Файл .ngd описує логічний дизайн, зведений до примітивних комірок пристрою Xilinx. Вихід у інструменті планування обладнання, що постачається із програмним забезпеченням Xilinx ISE. Тут

визначення обмежень - це не що інше, як присвоєння портів у дизайні фізичним елементам (наприклад перемикачам, кнопкам тощо) цільового пристрою та вказівці вимог часу в дизайні [25]. Ця інформація зберігається у файлі під назвою UCF (файл обмежень користувачів). Інструменти, які використовуються для створення або модифікації UCF - це PACE, редактор обмежень тощо.

- Мапінг: процес мапінгу запускається після завершення процесу перекладу. Мапінг відображає логічний дизайн, описаний у файлі NGD, до компонентів / примітивів (Slices / CLBs), присутніх у файлі NCD, створеному процесом Map, для розміщення та маршрутування дизайну на цільовий дизайн FPGA. У цьому процесі вся схема розподіляється на підблоки, щоб вони могли вміщуватися в логічні блоки FPGA. Логіка, визначена у вхідному файлі NGD, відображається в цільових елементах FPGA, тобто CLB та IOB, і генерується вихідний NCD (власний опис схеми), який зображує дизайн, відображений у FPGA.
- PAR-процес: Після мапінгу дизайн розміщується та направляється. Для цього процесу використовується програма "Місце і маршрут" (PAR). Процес місця та маршруту розміщує підблоки з картографічного процесу в логічні блоки відповідно до обмежень та з'єднує логічні блоки. Наприклад, якщо підблок розміщений у логічному блоці, який знаходиться дуже близько до штифта IO, то це може заощадити час, але це може вплинути на деякі інші обмеження. Таким чином, компроміс між усіма обмеженнями враховується місцем і маршрутом. Інструмент PAR приймає відображений файл NCD у якості вхідного сигналу і видає повністю вичерпаний файл NCD як вихід. Вихідний NCD-файл складається з інформації про маршрутизацію. Під час процесу розміщення підблоків розміщуються відповідно до логіки, але ці блоки не мають фізичної маршрутизації між ними, а також з колодками U0, але між ними є лише логічний зв'язок, який можна чітко

побачити за допомогою редактора FPGA відразу після місця процес закінчується. Ці логічні з'єднання показані щурячими мережами в редакторі FPGA. Потім запускається процес Route, який здійснює фізичні з'єднання між підблоками, розміщеними на FPGA. З'єднання здійснюються за допомогою матриць комутатора.

- Генерація біт-потоку: набір бінарних даних, які використовуються для програмування налаштованого логічного пристрою, найчастіше називають бітовим потоком, хоча це дещо оманливо, оскільки дані не більше бітово орієнтовані, ніж дані процесора набору інструкцій, і зазвичай немає "Потокове". Хоча в процесорі набору інструкцій дані конфігурації насправді безперервно передаються у внутрішні блоки, вони, як правило, завантажуються в налаштовану логічну систему лише один раз під час початкової фази настройки.
- Файл програмування генерується за допомогою запуску процесу "Створити файл програмування". Цей процес можна запустити після повного маршрутизації дизайну FPGA. Процес створення файлу програмування запускає BitGen, програму генерації бітових потоків Xilinx, для створення файлу бітового потоку (.BIT) або (.ISC) для конфігурації пристрою Xilinx. Потім пристрій FPGA налаштовується за допомогою .bit-файлу за допомогою методу сканування межі JTAG. Після того як пристрій Spartan буде налаштований на передбачуваний дизайн, його роботу перевіряють, застосовуючи різні входи.
- Функціональне моделювання: Моделювання після перекладу (функціональне) може бути виконане перед картографуванням конструкції. Цей процес моделювання дозволяє користувачеві переконатися, що ваша конструкція була синтезована правильно та будь-які відмінності через нижчий рівень абстракції можуть бути визначені.
- Статичний аналіз часу: можна провести такі типи статичного аналізу часу:

- Статичний аналіз часу після встановлення часу: результати хронометражу в процесі розміщення часу можуть бути проаналізовані. Процес аналізу статичної синхронізації (Post Post fit) відкриває вікно аналізатора синхронізації, яке дозволяє вам інтерактивно вибирати хронологічні шляхи у своєму дизайні для відстеження.
- Статичний аналіз часу після мапінгу: Ви можете проаналізувати результати синхронізації процесу мапінгу. Звіти про часовий графік повідомлення на карті можуть бути дуже корисними при оцінці продуктивності часу [25]. Хоча затримки маршруту не враховуються, логічна затримка може дати цінну інформацію про дизайн. Якщо логічна затримка становить значну частину (> 50%) від загальної допустимої затримки шляху, шлях може не відповідати вашим вимогам в часі, коли додаються затримки маршрутизації. Затримка маршрутизації зазвичай становить 45% до 65% від загальної затримки шляху. Визначаючи проблемний шлях, ви можете пом'якшити потенціал, перш ніж вкладати час на місце і маршрут. Ви можете переробити логічні шляхи, щоб використовувати менший рівень логіки, позначити шляхи для спеціалізованих ресурсів маршрутизації, перейти до більш швидкого пристрою або виділити більше часу для шляху. Якщо логіка лише затримок припадає набагато менше (> 35%), ніж загальна допустима затримка для шляху або обмеження часу, тоді програмне забезпечення для маршрутного маршруту може використовувати дуже низький рівень зусиль щодо розміщення. У цих випадках зменшення рівня зусиль дозволяє скоротити час виконання, все ще дотримуючись вимог продуктивності. [26]

- Тестування:Для програмованого пристрою ви просто запрограмуєте пристрій і негайно маєте свої прототипи. Потім ви несете відповідальність за розміщення цих прототипів у вашій системі та

визначити, що вся система насправді працює правильно. Якщо ви дотримувались процедури до цього моменту, велика ймовірність, що ваша система буде працювати нормально лише з незначними проблемами. Ці проблеми часто можна вирішити шляхом зміни системи або зміни системного програмного забезпечення. Ці проблеми потрібно перевірити та задокументувати, щоб їх можна було виправити під час наступної редакції чіпа. На даний момент необхідна інтеграція системи та тестування системи, щоб гарантувати, що всі частини системи працюють правильно. Коли мікросхеми вводяться у виробництво, необхідно провести певний тест на випал вашої системи, який постійно перевіряє вашу систему протягом певного тривалого часу.

4.2. Тестування модуля на ПЛІС

В даному підрозділі наведено результати тестових випробувань модуля на ПЛІС моделі Xilinx Virtex 6.

В таблиці 4.1 наведено порівняння апаратних затрат цільової ПЛІС при різних значеннях розрядності чисел формату з плаваючою комою.

Таблиця 4.1. Апаратні затрати для синтезу модуля

	16 бітів		24 біти		32 біти	
	Загалом	Використання	Загалом	Використання	Загалом	Використання
Кількість регістрів	16680	4%	23014	5%	29655	7%
Кількість буферів вводу-виводу	60	10%	80	14%	97	16%
Затримка	4,39нс		4,85нс		4,95нс	
Частота	227,30МГц		206,07МГц		202,03МГц	

ВИСНОВКИ ДО РОЗДІЛУ 4

Було проведено тестові випробування модуля на заданій моделі ПЛІС – Xilinx Virtex 6. Отримано таблицю для порівняння апаратних затрат синтезу модуля для різних значень розрядності чисел формату з плаваючою комою.

ВИСНОВКИ

1. В даній роботі було проаналізовано алгоритм для обчислення тригонометричних функцій CORDIC, а також встановлено різницю та відношення між форматами подання чисел у двійковому вигляді.
2. Було вирішено використати одну з версій алгоритму CORDIC як основу для організації конвеєрної обробки даних за алгоритмом, а також запропоновано алгоритм для перетворення чисел з одного формату в інший.
3. Розроблений модуль був описаний на мові програмування VHDL, було виконано перевірку та аналіз результатів моделювання, а також протестовано працездатність модуля на ПЛІС.

ПЕРЕЛІК ПОСИЛАНЬ

1. J. Volder, “The CORDIC Trigonometric Computing Technique,” IRE Transactions on Electronic Computers, vol. EC-8, no. 3, pp. 330-334, 1959.
2. J. Walther, “A Unified Algorithm for Elementary Functions,” Proceedings of Spring Joint Computer Conference, vol. 38, pp. 37 9-385, 1971.
3. J. Duprat and J. Muller, “The CORDIC Algorithm: New Results for Fast VLSI Implementation,” IEEE Transactions on Computers, vol. 42, no. 2, pp. 168-178, 1993.
4. P. Meher, J. Valls, T. Juang, K. Sridharan and K. Maharatna, “50 Years of CORDIC: Algorithms, Architectures, and Applications,” IEEE Transactions on Circuits and Systems, vol. 56, no. 9, pp. 1893-1907, 2009.
5. J. Cavallaro and F. Luk, “Architectures for a CORDIC SVD Processor,” Proceedings of International Society for Optical Engineering, vol. 698, pp. 45-53, 1987.
6. S. Abdulla, H. Nam, M. Dermot and J. Abraham, “A High Throughput FFT processor with no Multipliers,” IEEE International Conference on Computer Design, no. 10, pp. 485-490, 2009.
7. B. Parharni, Computer arithmetic, algorithms and hardware design, Oxford University Press, ed. 13‘, 2000.
8. J. Harrison, T. Kubaska, S. Story and P. Tang, “The Computation of Transcendental Functions on the IA-64 Architecture,” Intel Technology Journal, Q4, 1999.
9. J. Volder, “The Birth of CORDIC,” Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology, vol. 25, pp. 101-105, June 2000.
10. M. Ercegovac and L. Tomas, “Redundant and On—Line CORDIC: Application to Matrix Triangularization and SVD,” IEEE Transactions on Computers, vol. 39, pp. 725-740, 1990.
11. S. Wang and E. Swartzlander, “Critically Damped CORDIC Algorithm,” Proceedings of the 37th Midwest Symposium on Circuits and Systems, vol.

1, pp. 236-239, 1994.

12. Y. Hu and S. Naganathan, “An Angle Recoding Method for CORDIC Algorithm Implementation,” IEEE Transactions on Computers, vol. 42, no. 1, pp. 99-102,
13. DSP Guru [Електронний ресурс], – режим доступу: http://www.dspguru.com/info/faqs/CORDIC.htm.
14. C. Lee and P. R. Chang, “A Maximum Pipelined CORDIC Architecture for Robot Inverse Kinematics Computation,” Proceedings of the Japan-USA Symposium on Flexible Automation, Osaka, Japan, pp. 45-51, 1986.
15. R. Andraka, “A survey of CORDIC algorithm for FPGA based computers,” International Symposium on Field Programmable Gate Arrays, no. 2, pp. 191-200, 1998.
16. T. Vladimirova and H. Tiggler, “FPGA Implementation of sine and cosine generators Using the CORDIC Algorithm,” Surrey Space Center, University of Surrey, Guildford, Surrey. [Електронний ресурс]. – Режим доступу: http://klabs.org/richcontent/MAPLDCon99/Papers/A2_Vladimirova_P.pdf.
17. Wang, S. and Piuri, V., "A Unified View of CORDIC Processor Design", Application Specific Processors, Edited by Earl E.Swartzlander, Jr., Ch. 5, pp. 121-160, Kluwer Academic Press, November 1996.
18. Surnmanasena M.G.B “A scale factor correction scheme for CORDIC algorithm”, IEEE , August 2008.
19. Hamill R. Mccanny “Online CORDIC algorithm and VLSI architecture for implementating OR—array processor”, IEEE February 2000.
20. Rao, P.R. Chakrabarti “High performance compensation technique for radiX-4 CORDIC algorithm”, IEEE September 2002.
21. Boudabous A, Ghozzi, M. W. Asmondi “Implementation of hyperbolic function using CORDIC algorithm”, IEEE December 2004.
22. Meng Qian “Application of CORDIC algorithm to neutral networks VLSI design”, IEEE October 2006.
23. Ching Shing Wu , An Ye Wu “Modified vector rotation CORDIC algorithm

and its application to FFT” , IEEE may 2000.

24. Ching Shing Wu, An Yen Wu “Modified vector rotational CORDIC algorithm and architecture I. S. Walther, “A unified algorithm for elementary functions,” Spring Joint Computer Conference, pp. 37 9-3 85, 1971.
25. Y. H. Hu and S. Naganathan, “An angle recoding method for CORDIC algorithm implementation,” IEEE Trans. on Computers, vol. 42, pp. 99-102, Jan. 1993.
26. Y. H. Hu, “CORDIC-based VLSI architectures for digital signal processing” ,IEEE Signal processing Magazine, pp. 16-35, July 1992.

ДОДАТОК А. Опис модуля мовою VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

package cordic is
  component clk_gen is
       generic(
              clk_period : time := 10ns;
              angle_gain_coef : real := 200.0;
              SIZE       : positive := 32;
       ITERATIONS : positive := 16;
              RESET_ACTIVE_LEVEL : std_ulogic := '1'
              );
       port(
              CLK : out STD_ULOGIC;
              RST : out STD_ULOGIC;
              Angle : out std_logic_vector(SIZE-1 downto 0)
              );
       end component;

  component cordic_pipelined is
    generic (
       SIZE               : positive;
       ITERATIONS         : positive;
       RESET_ACTIVE_LEVEL : std_ulogic
    );
    port (
       Clock : in std_ulogic;
       Reset : in std_ulogic;
              Busy  : out std_ulogic;
              X : in signed(SIZE-1 downto 0);
       Y : in signed(SIZE-1 downto 0);
       Z : in signed(SIZE-1 downto 0);
              X_result : out signed(SIZE-1 downto 0);
       Y_result : out signed(SIZE-1 downto 0);
       Z_result : out signed(SIZE-1 downto 0)
    );
  end component;

  component sincos_pipelined is
    generic (
              MAGNITUDE  : real := 1.0;
       SIZE       : positive := 32;
       ITERATIONS : positive := 16;
       RESET_ACTIVE_LEVEL : std_ulogic := '1'
    );
    port (
       Sin   : out signed(SIZE-1 downto 0);
       Cos   : out signed(SIZE-1 downto 0)
    );
  end component;


  function cordic_gain(Iterations : positive) return real;

  procedure adjust_angle(X, Y, Z : in signed; signal Xa, Ya, Za : out signed);

  function float_to_signed(float_in: std_logic_vector(31 downto 0); amplitude : real) return
signed;
```

```
  function signed_to_float(signed_in : signed(31 downto 0); amplitude : real) return
std_logic_vector;
end package;




library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;


package body cordic is
  function cordic_gain(iterations : positive) return real is
    variable g : real := 1.0;
  begin
    for i in 0 to iterations-1 loop
      g := g * sqrt(1.0 + 2.0**(-2*i));
    end loop;
    return g;
  end function;

  procedure adjust_angle(x, y, z : in signed; signal xa, ya, za : out signed) is
    variable quad : unsigned(1 downto 0);
    variable zp : signed(z'length-1 downto 0) := z;
    variable yp : signed(y'length-1 downto 0) := y;
    variable xp : signed(x'length-1 downto 0) := x;
  begin
    quad := unsigned(zp(zp'high downto zp'high-1));
    if quad = 1 or quad = 2 then
      xp := -xp;
      yp := -yp;
      zp := (not zp(zp'left)) & zp(zp'left-1 downto 0);
    end if;
    xa <= xp;
    ya <= yp;
    za <= zp;
  end procedure;

  function float_to_signed(float_in: std_logic_vector(31 downto 0);
             amplitude : real) return signed is
     variable exp, mant : integer;
    variable mant_real, res_real : real;
    variable res : signed(31 downto 0);
  begin
      exp := to_integer(unsigned(float_in(30 downto 23))) - 127;
      mant := to_integer(unsigned('1' & float_in(22 downto 0)));
      mant_real := real(mant) / (2.0**23);
      res_real := mant_real * 2.0**exp;
      if float_in(31) = '1' then
          res_real := - res_real;
      end if;
        res := to_signed(integer((res_real / amplitude)
            * (2.0**31 - 1.0)), 32);
      return res;
  end function;

  function signed_to_float(signed_in : signed(31 downto 0);
             amplitude : real) return std_logic_vector is
      variable res : std_logic_vector(31 downto 0);
      variable sign : std_logic;
      variable exp : std_logic_vector(7 downto 0);
      variable mant : std_logic_vector(22 downto 0);
      variable in_abs: signed(31 downto 0);
      variable in_int, exp_int, mant_int : integer;
      variable mant_real : real;
```

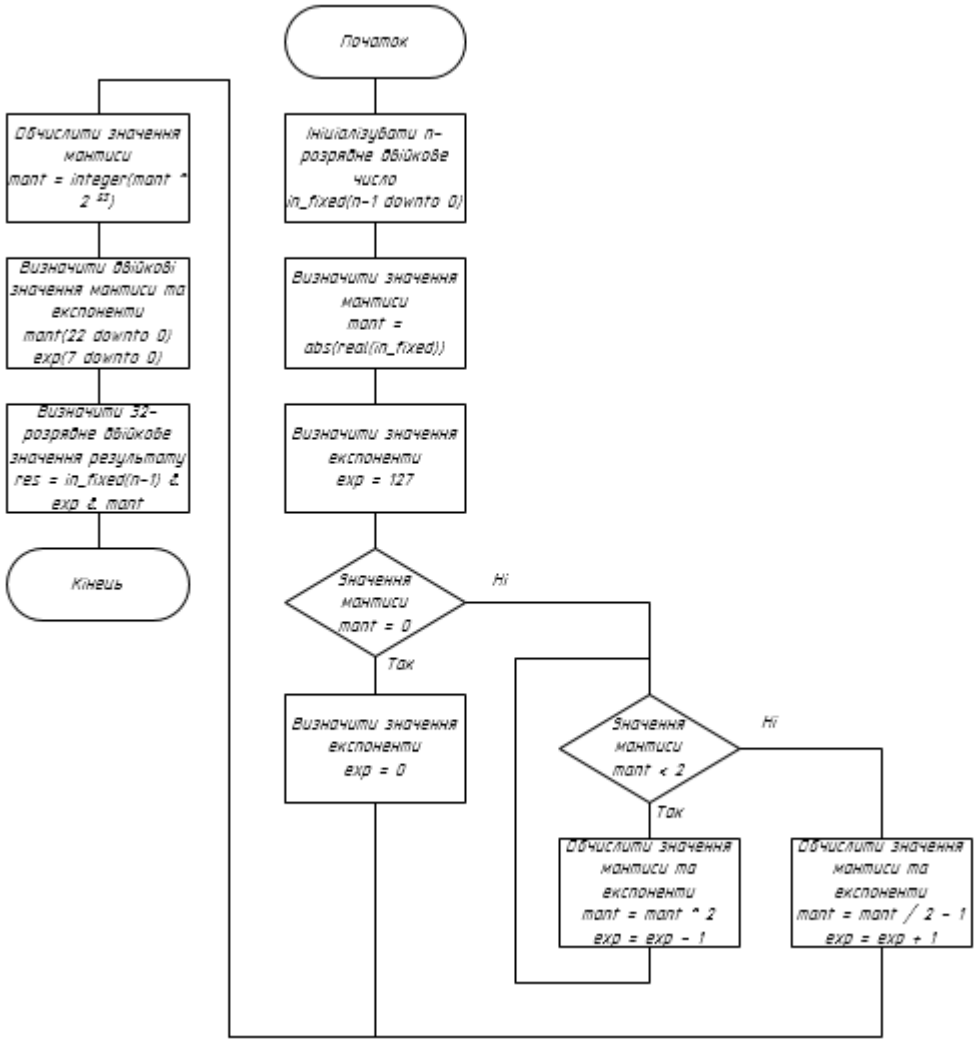
```
  begin
        sign := signed_in(31);
        in_abs := abs(signed_in);
        in_int := to_integer(unsigned(in_abs(30 downto 0)));
        mant_real := real(in_int) * amplitude / (2.0**31 - 1.0);
        exp_int := 127;
        if in_int /= 0 then
              while mant_real < 2.0 loop
                  mant_real := mant_real * 2.0;
                  exp_int := exp_int - 1;
              end loop;
              mant_real := mant_real / 2.0;
              exp_int := exp_int + 1;
              mant_int := integer((mant_real - 1.0) * (2.0**23));
        else exp_int := 0; mant_int := 0; sign := '0';
        end if;
        exp := std_logic_vector(to_unsigned(exp_int, 8));
        mant := std_logic_vector(to_unsigned(mant_int, 23));
        res := sign & exp & mant; return res;
  end function;
end package body;




library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

library BDP;
use BDP.cordic.all;


entity clk_gen is
generic(
              clk_period : time := 10ns;
              angle_gain_coef : real := 200.0;
              SIZE       : positive := 32;
       ITERATIONS : positive := 16;
              RESET_ACTIVE_LEVEL : std_ulogic := '1'
              );
port(
              CLK : out STD_ULOGIC;
              RST : out STD_ULOGIC;
              Angle : out std_logic_vector(SIZE-1 downto 0)
);
end entity;

architecture a of clk_gen is
       signal clk_loc: STD_ULOGIC;
       signal cur_iter : integer;
       signal angle_real : real;
       signal angle_signed : signed(SIZE-1 downto 0);
       signal angle_float : std_logic_vector(SIZE-1 downto 0);
begin
process begin
       RST <= '1';
       CLK <= '0';
       wait for clk_period;
       RST <= '0';
       CLK <= '1';
       clk_loc <= '1';
       cur_iter <= 0;
       angle_real <= - MATH_PI;
       loop
              wait for clk_period;
```

```
            cur_iter <= cur_iter + 1;
            clk_loc <= not clk_loc;
            CLK <= not clk_loc;
            angle_signed <= to_signed(integer( (angle_real / MATH_PI)
            * (2.0**(SIZE-1) - 1.0) ), SIZE);
            Angle <= signed_to_float(angle_signed, MATH_PI);
            if cur_iter = ITERATIONS * 4 + 4 then
                  RST <= '1';
                  cur_iter <= 0;
                  if integer((angle_real / MATH_PI) * angle_gain_coef)
                  < integer(angle_gain_coef) then
                        angle_real <= angle_real + (MATH_PI / angle_gain_coef);
                  else angle_real <= - MATH_PI;
                  end if;
            else RST <= '0';
            end if;
      end loop;
end process;
end architecture;




library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

library BDP;
use BDP.cordic.all;


entity cordic_pipelined is
  generic (
    SIZE       : positive;
    ITERATIONS : positive;
    RESET_ACTIVE_LEVEL : std_ulogic
  );
  port (
    Clock : in std_ulogic;
    Reset : in std_ulogic;
       Busy  : out std_ulogic;
       X : in signed(SIZE-1 downto 0);
    Y : in signed(SIZE-1 downto 0);
    Z : in signed(SIZE-1 downto 0);
       X_result : out signed(SIZE-1 downto 0);
    Y_result : out signed(SIZE-1 downto 0);
    Z_result : out signed(SIZE-1 downto 0)
  );
end entity;


architecture a of cordic_pipelined is
  type signed_pipeline is array(natural range <>) of signed(SIZE-1 downto 0);
  signal x_pl, y_pl, z_pl : signed_pipeline(1 to ITERATIONS);
  signal x_array, y_array, z_array : signed_pipeline(0 to ITERATIONS);
  signal cur_iter, cur_cycle : integer;

  function gen_atan_table(size : positive; iterations : positive) return signed_pipeline is
    variable table : signed_pipeline(0 to ITERATIONS-1);
  begin
    for i in table'range loop
      table(i) := to_signed(integer((arctan(2.0**(-i)) / MATH_PI) * (2.0**(SIZE-1) - 1.0)),
SIZE);
    end loop;
       return table;
  end function;
```

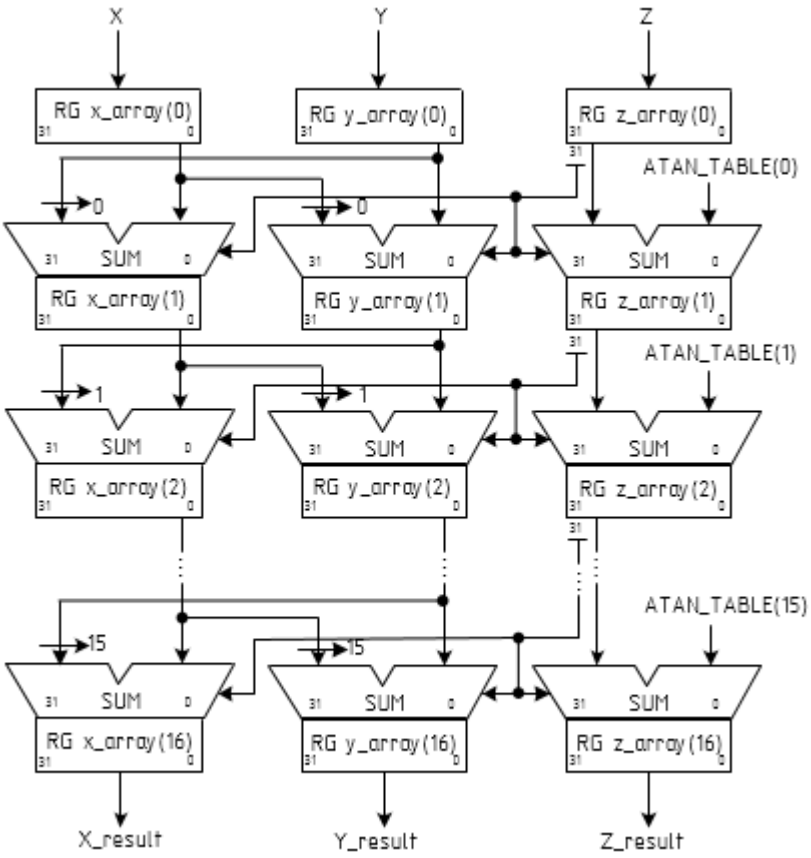
```
  constant ATAN_TABLE : signed_pipeline(0 to ITERATIONS-1) := gen_atan_table(SIZE,
ITERATIONS);
begin

  x_array <= X & x_pl;
  y_array <= Y & y_pl;
  z_array <= Z & z_pl;

  cordic: process(Clock, Reset) is
    variable negative : boolean;
  begin
    if Reset = RESET_ACTIVE_LEVEL then
        Busy <= '1';
        cur_cycle <= 1;
        cur_iter <= 1;
        x_pl <= (others => (others => '0'));
      y_pl <= (others => (others => '0'));
      z_pl <= (others => (others => '0'));

    elsif rising_edge(Clock) then
        if cur_iter /= ITERATIONS+1 then
          negative := z_array(cur_iter-1)(z'high) = '1';
             if negative then
          x_pl(cur_iter) <= x_array(cur_iter-1) + (y_array(cur_iter-1)
                    / 2**(cur_iter-1));
          y_pl(cur_iter) <= y_array(cur_iter-1) - (x_array(cur_iter-1)
                    / 2**(cur_iter-1));
          z_pl(cur_iter) <= z_array(cur_iter-1) + ATAN_TABLE(cur_iter-1);
        else
          x_pl(cur_iter) <= x_array(cur_iter-1) - (y_array(cur_iter-1)
                    / 2**(cur_iter-1));
          y_pl(cur_iter) <= y_array(cur_iter-1) + (x_array(cur_iter-1)
                    / 2**(cur_iter-1));
          z_pl(cur_iter) <= z_array(cur_iter-1) - ATAN_TABLE(cur_iter-1);
        end if;
            cur_iter <= cur_iter + 1;
        elsif cur_cycle = 2 then
               Busy <= '0';
               cur_iter <= ITERATIONS+1;
        else
               cur_cycle <= cur_cycle + 1;
               cur_iter <= 1;
        end if;
       end if;
  end process;

  X_result <= x_array(x_array'high);
  Y_result <= y_array(y_array'high);
  Z_result <= z_array(z_array'high);

end architecture;




library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

library BDP;
use BDP.cordic.all;


entity sincos_pipelined is
  generic (
```

```
      MAGNITUDE  : real := 1.0;
      SIZE       : positive := 32;
    ITERATIONS : positive := 16;
    RESET_ACTIVE_LEVEL : std_ulogic := '1'
  );
  port (
    --Sin   : out signed(SIZE-1 downto 0);
    --Cos   : out signed(SIZE-1 downto 0)
      Sin   : out std_logic_vector(SIZE-1 downto 0);
      Cos   : out std_logic_vector(SIZE-1 downto 0)
  );
end entity;


architecture a of sincos_pipelined is
  component clk_gen is
      generic(
            clk_period : time := 10ns;
            angle_gain_coef : real := 200.0;
            SIZE       : positive := 32;
      ITERATIONS : positive := 16;
            RESET_ACTIVE_LEVEL : std_ulogic := '1'
            );
      port(
            CLK : out STD_ULOGIC;
            RST : out STD_ULOGIC;
            Angle : out std_logic_vector(SIZE-1 downto 0)
            );
      end component;

  component cordic_pipelined is
    generic (
      SIZE               : positive;
      ITERATIONS         : positive;
      RESET_ACTIVE_LEVEL : std_ulogic
    );
    port (
      Clock : in std_ulogic;
      Reset : in std_ulogic;
            Busy  : out std_ulogic;
            X : in signed(SIZE-1 downto 0);
      Y : in signed(SIZE-1 downto 0);
      Z : in signed(SIZE-1 downto 0);
            X_result : out signed(SIZE-1 downto 0);
      Y_result : out signed(SIZE-1 downto 0);
      Z_result : out signed(SIZE-1 downto 0)
    );
  end component;

  signal Angle : std_logic_vector(SIZE-1 downto 0);
  signal Clock, Reset, Busy : std_ulogic;
  signal Angle_signed, Sin_loc, Cos_loc, Sin_loc_new, Cos_loc_new,
      xa, ya, za : signed(SIZE-1 downto 0);
begin

  U1:clk_gen
  generic map (
      SIZE => SIZE,
      ITERATIONS => ITERATIONS,
        RESET_ACTIVE_LEVEL => RESET_ACTIVE_LEVEL
  ) port map (
      CLK => Clock,
      RST => Reset,
      Angle => Angle
  );

  U2: cordic_pipelined
```

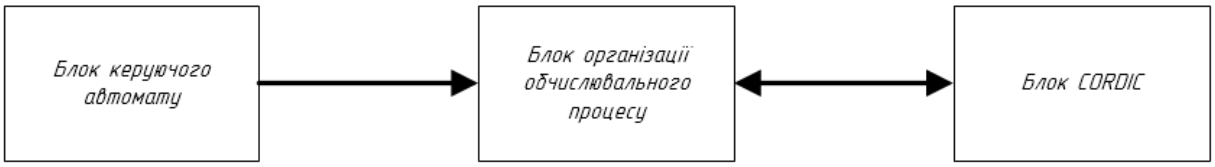
```
    generic map (
      SIZE => SIZE,
      ITERATIONS => ITERATIONS,
      RESET_ACTIVE_LEVEL => RESET_ACTIVE_LEVEL
    ) port map (
      Clock => Clock,
      Reset => Reset,
         Busy => Busy,
         X => xa,
      Y => ya,
      Z => za,
         X_result => Cos_loc_new,
      Y_result => Sin_loc_new,
         Z_result => open
    );

  P: process(Clock, Reset, Busy) is
    constant Y : signed(SIZE-1 downto 0) := (others => '0');
    constant X : signed(SIZE-1 downto 0) :=
        to_signed(integer((MAGNITUDE / cordic_gain(ITERATIONS))
          * (2.0**(SIZE-1) - 1.0)), SIZE);
     variable new_values : boolean;
  begin
      if Reset = RESET_ACTIVE_LEVEL then
      xa <= (others => '0');
      ya <= (others => '0');
      za <= (others => '0');
    elsif rising_edge(Clock) then
        Angle_signed <= float_to_signed(Angle, MATH_PI);
      adjust_angle(X, Y, Angle_signed, xa, ya, za);
    end if;

      new_values := Busy = '0';
        if new_values then
                   Sin_loc <= Sin_loc_new;
                   Cos_loc <= Cos_loc_new;
        end if;
  end process;

  --Sin <= (not Sin_loc(31)) & Sin_loc(30 downto 0);
  --Cos <= (not Cos_loc(31)) & Cos_loc(30 downto 0);
  Sin <= signed_to_float(Sin_loc, 1.0);
  Cos <= signed_to_float(Cos_loc, 1.0);

end architecture;
```



					ІАЛЦ.467400.004 Д1					
Зм.	Аркуш	№ Докум.	Підпис	Дата	Перетворення числа з фіксованою комою в число з плаваючою комою. Блок-схема алгоритму					
Розробив		Петрик Р.І.								
Перевірив		Сергієнко А.М.								
						Аркуш 1		Аркушів 1		
Н. контр.		Сімоненко В.П.			Дипломний проєкт	НТУУ "КПІ", ФІОТ, ІО-61				
Затвердив		Стіренко С.Г.								



					ІАЛЦ.467400.005 Д2					
Зм.	*Аркуш*	*№ Докум.*	*Підпис*	*Дата*	*Конвеєризований CORDIC. Схема електрична функціональна*					
Розробив		*Петрик Р.І.*								
Перевірив		*Сергієнко А.М.*								
						Аркуш 1		*Аркушів 1*		
Н. контр.		*Сімоненко В.П.*			*Дипломний проєкт*	*НТУУ "КПІ", ФІОТ, ІО-61*				
Затвердив		*Стіренко С.Г.*								



					ІАЛЦ.467400.006 Д3					
					Модуль для обчислення тригонометричних функцій з плаваючою комою. Схема електрична структурна					
Зм.	Аркуш	№ Докум.	Підпис	Дата						
Розробив		Петрик Р.І.								
Перевірив		Сергієнко А.М.				Аркуш 1		Аркушів 1		
Н. контр.		Сімоненко В.П.			Дипломний проєкт	НТУУ "КПІ", ФІОТ, ІО-61				
Затвердив		Стіренко С.Г.								